MIPS® SDE 6.x Programmers' Guide

Document Number: MD00428
Revision 1.17
April 4, 2007

MIPS Technologies, Inc
1225 Charleston Road
Mountain View, CA 94043-1353

Template: B1.15, Built with tags: 2B MIPS32/MIPS64 SUM

Open Source Copyright Notices

Many of the utilities contained in this package are derived from Free Software Foundation code, which require this notice:

> This program is free software; you can redistribute it and/or modify it
> under the terms of the GNU General Public License as published by the
> Free Software Foundation; either version 1, or (at your option) any
> later version.
>
> This program is distributed in the hope that it will be useful,
> but WITHOUT ANY WARRANTY; without even the implied warranty of
> MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
> GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License
> along with this program; if not, write to the Free Software
> Foundation, 675 Mass Ave, Cambridge, MA 02139, USA.

See also http://www.fsf.org/licenses/licenses.html.

The Windows version of MIPS® SDE is built using the *Cygwin* programming environment, produced by RedHat/Cygnus at http://www.cygwin.com, and is distributed under the terms of the GNU General Public License and Cygwin API license, see http://cygwin.com/licensing.html.

Some of the target libraries contain modules which require this acknowledgement:

> This product includes software developed by the University of
> California, Berkeley and its contributors.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Ltd. X/Open is a trademark of X/Open Company Ltd. POSIX is a registered trademark of the Institute of Electrical and Electronics Engineers, Inc. ThreadX® is a registered trademark of Express Logic, Inc.

For more details on how these copyrights affect the code that you develop using MIPS® SDE, see Appendix A "Copyrights".

Table of Contents

Figures

Tables

Introduction

This is a programmers' guide for MIPS Technologies' Software Development Environment for MIPS-Based™ products (henceforth just called "SDE" in this manual).

SDE is a software engineer's cross-development system for MIPS architecture processors, intended for statically-linked embedded applications running on "bare metal" CPUs or light-weight operating systems[1]. It is a component of the MIPS® Software Toolkit (henceforth "MTK"), which includes not only SDE, but other tools and libraries intended to accelerate the development of high quality, high performance applications running on MIPS Technologies' cores. Another key component of MTK is the powerful MIPSsim™ simulator. Expect to see additional components being added to MTK in future releases.

This manual describes the supported version of MIPS® SDE included in the MIPS® Software Toolkit, as well as a freely downloadable, but unsupported subset called MIPS® SDE *lite*.

SDE provides much more than just prebuilt GNU binaries; it has everything that you need to build and debug downloadable and, for some targets, standalone rommable code (including MIPS-specific low-level CPU initialization and management). It is hosted on Windows (NT, 2000 and XP), Linux/x86, Linux/AMD-64, and Sun Solaris.

What's in MIPS® SDE 6.x

SDE is built around GNU tools tuned, enhanced and packaged by MIPS Technologies together with a set of C and C++ libraries, and a workable single-tasking run-time system. It is maintained independently, by which we mean we will never tell a supported customer that they need to wait while someone else fixes something – and, implicitly, that we maintain our own independently tested codebase.

The SDE run-time system includes convenient C interfaces to pretty much every strange thing you'll have to program on a MIPS-Based processor. At a higher level it conforms closely to POSIX standards – so if you need to port your software to or from other operating systems or CPUs, then there's a road open.

SDE is command-line based, and if your background is with PC "integrated development environments" that may come as a culture shock. But stay with us; there's certainly a lot to learn about tools of this kind, but most of it can be learned while you are doing useful work. If you're not quite up to speed on command-lines, read Chapter 6 "Overview". Then read the rest of this page for some useful jumping-off points into the rest of this manual.

The SDE toolkit is structured around a number of example programs, each of which can be built out of the box for the simulators we include, or for any of the supported evaluation boards. You will be going with the flow if you try one of the examples first, and pick one of the examples as a template for any software you want to port to the MIPS architecture.

The MIPS® SDE *lite* subset

The GNU tools themselves are freely redistributable software, and MIPS Technologies provides a free-to-download subset of SDE, called SDE *lite*. It has the same features as the full version, but the proprietary run-time software is provided only as precompiled libraries, not as reusable source code. More important: the free version does not come with support. For more information about your rights and obligations regarding the use of derived binaries see Appendix B "MIPS™ Freedom-to-Use License". But if you've used the free version, like what you've seen so far, and want to upgrade to the full, supported version, then see Chapter 24 "Getting Support". From now on we'll normally just say "SDE" when we mean either the full SDE, or SDE *lite* .

Getting working fast

To get started right away, first follow the installation instructions from Chapter 3 "Installation", and then proceed straight to Chapter 5 "Quick Start", which shows how to run the simplest possible program on the easiest possible MIPS-Based target – a software simulator supplied with SDE.

[1] We also have a version of our toolchain configured as a Linux/MIPS native compiler, generating MIPS/abi PIC code, but this manual does not describe that – instead see [MD00410].

If your priority is to run some particular programs – perhaps benchmarks – on one of the evaluation boards or simulators (“targets”) supported by SDE, then the next thing to do is to build the support library for your target, as described in Chapter 8 “Target Specific Libraries”. You can then try running one or both of the benchmark examples (*dhrystone* and *whetstone*) provided with SDE: see Section 9.1.5 “Dhrystone Benchmark”.

If you have any problems compiling your own benchmark – and certainly before you tell anyone else the results – you should read Chapter 10 “Porting an ISO / ANSI C Program”, which warns of potential portability problems. If that’s not enough, then Chapter 14 “Debugging with GDB” shows you how to connect the source-level debugger to your target and find out what’s going wrong. And read Section 12.2 “Optimization Options” and Chapter 15 “Profiling with GPROF and GCOV” to see how you can improve your results.

If you are developing or porting a more complex program that needs low-level access to the hardware, then SDE also provides some viable and robust run-time components. Read Chapter 19 “SDE Run-time I/O System” for a description of the programmer’s interface to the CPU management functions.

If you need to study or modify the run-time system and CPU management source code, then refer to Chapter 21 “Embedded System Kit Source ”, which is a guide to its structure. If you want to run programs on a board or other target which is not already supported by SDE, then you will have to write some new board-specific code. Chapter 22 “Retargetting the Toolkit” tells you how you can save effort by writing your board support code the SDE way. In either case, you’ll need more source code than is provided in the SDE *lite* subset – you’ll need to have the supported MIPS® Software Toolkit.

Throughout most of this manual we’ll show file locations relative to the directory where you install SDE by starting them off with three dots (an ellipsis) and using UNIX-style forward slashes, like this: `.../sde/examples`. See Section 2.1.1 “File pathnames in Windows with Cygwin” and Section 3.3 “Installation” for more details.

Other reading

In Chapter 25 “References” at the end of the manual you’ll find details of other books we’ve found helpful. But two in particular are worth getting at this stage:

- To understand what makes the MIPS architecture different, get used to the MIPS buzzwords, and feel some comfort with MIPS programming at the assembly language level you should read *See MIPS Run* [Sweet99][2]
- If you’re going to use SDE’s libraries and run-time system it’s worth getting hold of the *POSIX Programmer’s Guide* [Lewine91].

In fact, this may be a good time to take a quick look at Chapter 25 “References” and run up a bill at your local computing bookshop.

Other toolchain documentation

The individual GNU tools which make up so much of SDE have individual generic manuals: [Binutils], [Cpp], [Gcc], [Gdb], [Gprof], [Ld], [Make], [Stabs].

The manuals are extensive, very detailed and cover many different CPU types; many are very well-written and are an excellent, but not fast, read. We don’t include printed versions with our software package, but you will have HTML versions you can read on-line with your web browser as described in Chapter 7 “Online Documentation”, and PDF versions you can print out for yourself.

Other components of the MIPS® Software Toolkit package come with their own detailed manuals.

[2] The square brackets tell you that this is a reference to another publication, listed in Chapter 25 “References”.

SDE on UNIX and Windows

While SDE runs well on Windows systems, its origins were on UNIX. SDE is ported to Windows using the "Cygwin" system, as described in this chapter, and Cygwin supports both Windows pathnames (with back-slashes) and UNIX-style file pathnames with forward-slashes. As supplied all SDE's build examples are written with UNIX-style pathnames; so the following sections explain the important issues for Windows users.

2.1 SDE on Windows and "Cygwin"

SDE tools are real 32-bit Windows applications, but apart from the debugger they're command-line programs most easily launched from a console window; that might be from inside the debugger, a programmer's editor, or the UNIX-like Cygwin command-line "shell" window.

If this is new to you don't panic yet: you rarely need to type a command more complicated than "*sde−make something*", unless you get to like command lines. Windows users are likely to wrap the command line tools up using a commercial programmer's editor, browser or "IDE" product. Most of the popular compiler-independent front-ends are readily configured around GNU tools.

To keep the sources as similar as possible, the version for Windows is built using the "Cygwin" DLL[3]. Cygwin offers a POSIX[4]-compatible API for Windows, allowing us to build UNIX and Win32 versions of software from the same sources, with relatively few system dependencies.

The Cygwin DLL is accompanied by a package of GNU command line utility programs. They're widely used by "makefiles" which co-ordinate software builds, so are invaluable to those wanting to port a build process from a UNIX to a Windows host. In particular, quite a few of them are used by the SDE makefiles.

> The Windows release of SDE v6.06 requires the user to install Cygwin first, then install SDE tools using Cygwin facilities. This may change in a future release.

Customers with an active support or maintenance contract with MIPS Technologies can receive support for those Cygwin utilities which are used in our makefiles; any problem with those should be reported and we'll fix them. The Cygwin GNU utilities *not* used in our makefiles are "contributed software" and we don't guarantee to tackle bugs in them.

2.1.1 File pathnames in Windows with Cygwin

UNIX and the world-wide Web use forward slashes "/" to separate the components of pathnames; when MS−DOS introduced pathnames they used back-slashes "\"[5], and Windows has kept to that. Moreover, full MS−DOS pathnames start with a drive letter such as "`C:`".

When you use SDE on Windows (courtesy of Cygwin) either pathname format can be used. That doesn't make them equally usable in all cases. For general file system purposes you'll probably tend to use Windows navigation tools, but Cygwin's UNIX-derived applications make large-scale use of backslash as an escape character and you'll struggle to sneak backslashes past UNIX-style command and option parsers. Similar problems are caused by spaces in filenames, and the MS−DOS "`x:`" syntax can cause confusion in UNIX search paths, which use '`:`' as a pathname separator (where MS−DOS and Windows use '`;`').

If SDE users hit problems, it will probably be in *makefiles*. Let us know what happens and we'll try to fix it. The exact relationship between Windows and Cygwin pathnames depends on settings in the Windows "Registry", but in most cases all the following are equivalent:

[3] Many thanks are due to Cygnus Solutions (now part of Red Hat, `http://www.redhat.com`), whose staff carried out this work and opened up Win32 environments to GNU and other freely redistributable software.

[4] "POSIX" is a set of standards to allow software portability across a very large range of computer systems, which grew up in the UNIX world.

[5] It probably wasn't *just* perversity; MS−DOS applications had already fixed on "/" to mark command line options.

```
c:\Windows\System
c:/Windows/System
//c/Windows/System
/cygdrive/c/Windows/System
```

The further down that list you go, the more compatible you'll be with UNIX-style command and option parsers. Definitely don't expect to get away with spaces, dollar signs, or parentheses in filenames inside a makefile.

Cygwin uses a mapping table called the *mount table*, stored in the Windows registry, to allow Windows drive names to appear as a single, unified POSIX file system. The mount table concept will be familiar to many UNIX users, but old DOS hands may also recognise it as similar to the `join` command, which made individual drives appear to be part of a single file tree. The mount table is manipulated by Cygwin's `mount` and `umount` commands. The `cygpath` command can convert between POSIX and Windows file name formats, in case you need to do that in a "shell script", batch file or *makefile*.

Remote network shares can be accessed directly using the "UNC" `//servername/sharename` convention – they don't have to be *mounted* first.

A more detailed description of how Cygwin file naming and the mount table works can be found at http://cygwin.com/cygwin-ug-net/using.html.

2.1.2 Text and binary files in Cygwin

Another major schism between the Windows and UNIX world is the convention on how to mark the end of a line in a text file: UNIX programs use a single line-feed character (ASCII LF), while Windows uses a carriage-return, line-feed pair (ASCII CR/LF) and an ASCII SUB (Control-Z) to indicate end-of-file. Therefore on Windows a C program must indicate whether it is writing to a file in text or binary mode, which tells the i/o libraries whether to expand '`\n`' to CR/LF when writing a file – and vice versa when reading. This is true of Cygwin programs too, but with Cygwin you can control whether this translation occurs on a "per mount-point" basis using the `mount` command's **–b** (binary) or **–t** (text) option: in a binary mode file system text and binary files are treated identically, i.e. no translation is done and UNIX-style single LF line endings will be written to output files, and expected on input files; in text mode file systems the text conversion is performed.

The choice of which file system mode to use probably depends on the editor you are going to use with your source files. If you use a Cygwin-based text editor (e.g. XEmacs, Emacs, vi, nano, ed), then you'll do best with binary mode. If you already use a Windows program editor which can't be instructed to use UNIX line endings, then you'll do better selecting text mode. In desperation the Windows *WordPad* editor understands UNIX line endings, and may be acceptable for occasional usage – it can be called up from the command line using the `write` command, for example:

```
$ cd .../sde/examples/hello
$ write hello.c
```

If you need to convert text files between UNIX and DOS line endings, you can use the `unix2dos` and `dos2unix` utilities, supplied as part of the optional *cygutils* package[6]. For example, SDE source and headers are supplied in UNIX format, so the following command line run in a Cygwin shell window would convert all of SDE's text files from UNIX to DOS line endings:

```
$ cd .../sde/
$ find kit include examples -type f \! -name "*.lib" | xargs unix2dos
```

2.2 Environment variables

Environment variables are used in both UNIX and Windows; the best-known is the PATH variable, which specifies a list of directories to search for programs[7].

[6] Use the Cygwin Setup program to install the *cygutils* package – it's in the "Utils" category.

[7] Unix users may be surprised that the current directory is implicitly searched first for executables, even if it is not listed in the Windows PATH variable.

Each variable is just a name and associated string value. Whenever one program launches another, all these names and values are copied to the "child" program. By means of that inheritance, the variables are useful for defining global "facts" about the way you use the system which different programs can use to fit in with it; in particular the "*sde−make*" program which orchestrates software builds under SDE uses environment variables to define build rules.

Variables are most usually initialised by running a script which uses one of several flavours of "set variable" command. In UNIX systems the variables are typically set up by your login or your personal command-line shell startup script, so your environment settings depend on your log-in identity. For this purpose Cygwin creates a UNIX-compatible user id and home directory on Windows NT and above – by default that will be "`/home/username`".

When you install the software on Linux or Windows you'll get a choice between making the software available to all users[8], and making it available just for you. On Solaris it's probably just for you.

[8] It relies on the convention that all users' shell interpreters execute the scripts in directory `/etc/profile.d/` when they start up. Both Cygwin and many modern Linux distributions will do that, but on Linux you will need to have "super-user" privileges to be able to create files in that directory.

Installation

Whatever else you skip, please read this section...

3.1 Minimum System Requirements

- *Platform* : Any of the following hosts, running one of these named operating systems (of **at least** the specified version number):
  - x86 – Microsoft Windows NT, 2000, XP, with Cygwin 1.5.11 or above.
  - x86 – RedHat Linux 7.1 or higher, but pretty much any x86 Linux with `glibc` version 2.2.3 or higher should be OK.
  - AMD-64 – RedHat Linux with `glibc` version 2.3.2 or higher.
  - SPARC – Solaris 2.6 or higher.

  If you've got some flavour of UNIX or Windows which isn't on this list and can't be supported by any of the above, please ask or we won't know we're missing you.
- *Memory* : 64Mbytes should be fine for most purposes, but nowadays you'll probably have much more than that.
- *Disk Space* : 500 Mbytes available.

3.2 Environment Variable Setup

The SDE installation process gives the choice to modify the PATH environment variable (making SDE tools directly usable to you) by arranging to run the appropriate *sdeenv* script[9] whenever you start a shell. It uses two approaches, depending on your install-time choice:

- *For all users* : installs copieds of the *sdeenv* files in the `/etc/profile.d/` directory, where they will be executed automatically for every user.
- *Just for you* : adds a line to the end of your personal shell startup script (`.profile`, `.cshrc`, or `.tcshrc`) which invokes the appropriate *sdeenv* file.

With SDE v6.06 and above, running the tools from a DOS box or Windows "Run" dialog is possible, but is deprecated – you'd have to find your own way of setting the PATH variable and other Cygwin environment variables.

3.3 Installation

You should download SDE from the internet: you'll generally find the most recent recommended version at `http://www.mips.com`, and follow links to "Products" and "Software Tools"

Installation is "semi-automatic", using scripts. It usually works first time, but you should read these notes through before you start and take a little bit more trouble than you might with other software; SDE has hundreds of users, not tens of thousands, so now and again someone will come up against some configuration problem that we've never heard of before.

When you're downloading from internet you'll first obtain the SDE *lite* subset. If you purchased the MIPS® Software Toolkit you'll then receive additional components which extend this to form the full MTK version (you can download these using a login name and password we'll send you, or we can email them to you).

What's in the internet download?

The toolchain is provided as a gzip-compressed *tar*[10] archive, sometimes called a "tarball" for short. There is a single tar file for each supported host type, with a name like `PN00115-`*`xx.yy`*`-2B-MIPSSW-?SDE-v`*`a.b.c`*`.tgz`. This contains the GNU tools and documentation, plus MIPS Technologies' proprietary examples, libraries, header

[9] It can be `.../bin/sdeenv.sh` or `.../bin/sdeenv.csh`, depending on your choice of shell.

[10] The *tar* format is familiar to UNIX users, but many Windows packages (including freeware or shareware) can read it. Its virtue is its simplicity.

files, and run-time system. The "`?`" in the archive file name represents the host type, and the "`xx.yy`" and "`a.b.c`" strings are numeric sequences which encode the release number in a reasonably obvious way.

In addition to the per-host tar archive you'll also find some files which are for your information only:

- *README.TXT*: a plain text file, where we document any late updates to the release. It's the final authority about how to go about downloading, and might tell you of errors in or changes to this chapter, so read it.
- *NEWS*: a text file containing the recent release history.
- *sde-guide.pdf*:
- *MD00428-2B-SDE-SUM-xx.yy.pdf*: two different names for the latest version of this manual.
- *PN00119-xx.yy-2B-MIPSSW-SDE-SRC-va.b.c.tbz*: optional source code for the GNU programs, as a *bzip2*-compressed tarball – only serious hackers need this. If you need it, it is freely downloadable from the web site.

Most of these files are packaged for delivery inside yet another "meta" tarball, with a name like `IPDP00298-xx.yy-1D-MIPSSW-SDE-HOST-va.b.c-LITE.tgz`.

Where should you install your package?

In this manual we'll often refer to file pathnames. It would fatten the manual horribly to write them all twice (in Windows and UNIX format); so we'll most often just write them with forward slashes, as used on UNIX, in the Cygwin shell or the makefiles. When you're using native Windows tools, replace each "/" with a "\" and prepend the root of the Cygwin POSIX tree (e.g. `c:\cygwin\`).

SDE has a default location which it will search in for include files, libraries, etc: `/usr/local/sde6`. But nowadays this location is not compulsory – the tools will automatically find these files relative to their installed location. Wherever you choose to install SDE we'll call this the "*SDE root*", and all the files which make up the release will live in subdirectories below this point. In the remainder of this manual we'll write a pathname relative to the SDE root by starting it off with three dots (an ellipsis) like this: `.../`

> **Warning:** DO NOT install SDE in the Windows "`\Program Files`" directory – or anywhere else where there will be spaces in the pathname. Spaces in the pathname will be seen as separators on every command line or makefile line; it could be worked around, but all the standard makefiles will stop working.

The next sections tell you how to install the package on UNIX or Windows, from the internet – skip the sections you don't need. Do us and yourself a favour; read through to the end of this list before you start, so you get advance warning when we ask you to do something impossible.

Once you've completed the installation you can proceed to Chapter 5 "Quick Start" to try it out.

Install MIPSsim™ simulator and probes

If you purchased the MIPS® Software Toolkit, then you will have received a copy of the MIPSsim simulator. You may also have purchased a hardware EJTAG probe. In both cases we recommend that you install these tools first – before installing SDE – following the instructions supplied with these products. This will allow the SDE installation scripts to automatically configure the debugger to use your simulator and/or probe.

If you install these tools later – don't worry – you'll just have to teach SDE about them manually. Details about installing and using the MIPSsim simulator and EJTAG probes are in Section 14.1 "MDI Debugging".

Remove old SDE

Don't try to install a new major SDE release on top of an old one. Reorganizations between major releases of SDE are usually substantial enough that it is not possible to merge releases in this way. You must install SDE into a different directory. It is usually safe to install minor revision updates and patches on top of the same major release.

Consult Appendix C "Release History" for details of significant changes since the last release.

Windows: Uninstall old SDE and/or Cygwin

Recent versions of Cygwin have a structure which has changed so much that it is not usually safe to install them side-by-side with older Cygwin releases such as B18, B19, B20, B20.1 or 1.0. You should be better off with a new installation in any case. SDE v4.1 and earlier were built on Cygwin B20.1 or B19, so if you're upgrading from one of those releases you'll first have to uninstall your old copy.

To delete SDE v4.0 and above, choose *Remove Algorithmics Free GNU Toolkit* from the *Free GNU Toolkit* folder in your Windows *Programs* menu. You may then need to manually delete or rename any shortcuts to the old release from your Windows desktop.

UNIX/Linux: Uninstall old SDE

When removing an old SDE installation from a UNIX host you may need to identify and remove any SDE related changes to your `.cshrc`, `.tcshrc`, `.login` or `.profile` startup files and remove them. For releases prior to SDE v4.0 this means removing definitions of environment variables like `GCC_EXEC_PATH` and `LIBRARY_PATH`, which are no longer required, and would confuse the new tools.

Windows: Install Cygwin

Go to `http://www.cygwin.com` and follow the *Install Now!* link. Even if you've already got a recent net release of Cygwin installed, you must still follow these instructions to download the latest updates, and make sure that you are running Cygwin 1.5.11 or above.

When you run the downloaded Cygwin Setup program, one of the first dialog boxes is called "Select Root Install Directory", and it asks you three somewhat confusing questions:

- *Root Directory* : This Windows drive and directory is where the whole of the Cygwin pseudo-POSIX file system will be rooted. If you've had an old version of Cygwin (prior to version 1), such as the one included with SDE 4.x, then this will probably indicate the root of a Windows drive, e.g. "`C:\`". This is no longer recommended practice for Cygwin – instead you should install it in its own sub-directory, to avoid muddling its files up with other Windows programs. We recommend that you edit this field to read "`c:\cygwin`", or similar.
- *Install For: All Users / Just Me* : The Cygwin package uses the Windows registry to store its *mount table*, which it uses to map Windows drives and network shares into Cygwin's unified POSIX file tree. See Section 2.1.1 "File pathnames in Windows with Cygwin" for more details. If you select "All Users" then the Setup program will initialise the "system wide" mount table, shared by all users on this system; desktop and start menu shortcuts will also be created for all users. The "Just Me" option creates the mount table and desktop shortcuts only for the current user.
- *Default Text File Type: DOS / UNIX* : Selects the type of line endings in text files read or written by Cygwin programs. Cygwin defaults to "UNIX" mode, as this creates less problems for programs ported from UNIX, and it's faster – but it may not be the right choice for you if you are going to use Windows native text/program editors, in which case you should select "DOS" mode. See Section 2.1.2 "Text and binary files in Cygwin" for more discussion of this issue.

If you're new to Cygwin, the next most confusing choice you'll encounter will be what packages to install. The first time the Setup program is used it will select all packages in the "Base" category, and this is a sufficient minimum to run SDE. But there's lots more interesting software. You might want to add the `cygutils` package, part of the "Utils" category, which contains the text file conversion tools mentioned in Section 2.1.2 "Text and binary files in Cygwin".

The Setup program can be run again at any time to check for updates to your currently installed packages, or to download and install new contributed packages.

To choose packages from the ‘‘Select Packages’’ list:

1) First make sure that the ‘‘Curr’’ button is selected, not ‘‘Prev’’ or ‘‘Exp’’. The ‘‘Exp’’ button selects experimental (beta) releases, which are not recommended for production use.

2) You can use the ‘‘View’’ button to cycle between three views of the package list:

   - *Category* : a list of packages grouped by category, which sometimes make it easier to browse the list and find useful packages.
   - *All* : a complete list of all available packages, in alphabetical order.
   - *Partial* : a list of all packages currently selected for downloading and installation – when running Setup after the initial installation this will list available updates to your currently installed packages, if any.

3) In both the ‘‘Partial’’ and ‘‘All’’ views, each package shows the currently installed version (if any), and then an embedded ‘‘spinner’’ button. This button selects the action that will be performed to this package when you finally hit the ‘‘Next>’’ button. The possible states are:

   - *Skip* : this package is not currently installed – and will continue to be so.
   - *Keep* : this package is installed, but keep the current version – don’t update it.
   - *Uninstall* : remove this package.
   - *Reinstall* : download and reinstall the same version of this package as is already installed.
   - *version-number* : A newer, possibly experimental (beta) version of this package exists. Don’t select this option, unless you are doing an update run and Setup inserts this for you automatically in the ‘‘Partial’’ view, because it is a ‘‘current’’ update.

4) Finally, press the ‘‘Next’’ button to start the download and installation.

Once your Cygwin installation has finished, open a Cygwin ‘‘shell window’’ by activating your new Cygwin desktop icon, or start menu item.

If the shell prompt looks something like this:

```
Administrator@PCNAME $
```

or if the `id` command says that your name is ‘‘Administrator’’, then you need to update Cygwin’s `/etc/passwd` and `/etc/group` files, as follows:

```
$ mkpasswd -l -d | sort -u >/etc/passwd
$ mkgroup -l -d | sort -u >/etc/group
```

Then close your Cygwin window and open a new one. You should now see your Windows login name as part of your prompt.

By default Cywgin will inherit your network ‘‘home’’ directory from a Windows domain server, if there is one. If this isn’t what you want – you may prefer for speed or connectivity reasons to keep your Cygwin files only on the local machine – then you will need to edit the Cygwin `/etc/passwd` and change your home directory to be in the Cygwin local file system, e.g. `...//home/jdoe`. Cygwin will then create that directory when you next open a Cygwin shell window. The records in the `/etc/passwd` file consist of a colon-separated list of fields: find the line whose first fields starts with your login name, and change the last but one field to `/home/username`.

Now that you have Cygwin up and running, work inside a Cygwin shell window to install SDE *lite* or SDE: the instructions are now the same as for a UNIX installation.

Install SDE

1) You can either download SDE *lite* from http://www.mips.com, and follow links to "Products" and "Software Tools" or you may receive a copy on a CDROM. In either case you will either receive or download one or more compressed *tar* files, with names starting "IPD".

2) On Windows: open a Cygwin shell window.

3) The file(s) which you downloaded must first be unpacked using the *tar* command. For example:

```
$ cd /tmp
$ gzip -dc IPDP00298-01.00-1D-MIPSSW-SDE-LIN-v6.06.01-LITE.tgz | tar xf -
```

> Windows users might be tempted to unpack the *tgz* files using a program like WinZip or UltimateZip, but they will **not** work correctly – these programs don't understand how to handle symbolic links and other UNIX features that may be present. You must use the Cygwin *tar* command, as shown above.

This will leave you with one or more new tar files, named as follows:

Table 3-1 Installable tar files

Component	Purpose
PN00114-xx.yy-2B-MIPSSW-SSDE-va.b.c.tgz	Sparc Solaris host toolchain
PN00115-xx.yy-2B-MIPSSW-LSDE-va.b.c.tgz	x86 Linux host toolchain
PN00116-xx.yy-2B-MIPSSW-MSDE-va.b.c.tgz	Microsoft Windows host toolchain
PN00118-xx.yy-1C-MIPSSW-MTK-SDE-va.b.c.tgz	Extra MTK source code

4) Unpack the appropriate host toolchain tar file into your chosen SDE root directory, for example:

```
$ mkdir ~/sde-6.06
$ cd ~/sde-6.06
$ gzip -dc /tmp/PN00115-6.61-2B-MIPSSW-LSDE-v6.06.01.tgz | tar xf -
```

The '~' in the example is expanded by the shell to the name of your home directory. Do **not** use *WinZip* or any other native Windows program to unpack these files.

5) If you purchased the MIPS® Software Toolkit then you should now unpack the additional PN00118-*xx.yy*-1C-MIPSSW-MTK-SDE-va.*b.c*.tgz archive which you received from us into the same SDE root directory. It contains the extra components which upgrade SDE *lite* to the supported MTK version of SDE. For example:

```
$ cd ~/sde-6.06
$ gzip -dc /tmp/PN00118-6.61-1C-MIPSSW-MTK-SDE-v6.06.01.tgz | tar xf -
```

6) Run the setup script, e.g.:

```
$ sh ./bin/sdesetup.sh
```

This will auto-generate the startup scripts which add the SDE tools to your search path. It will also ask you if you wish to configure one or more "MDI fragments" – configuration files which tell the *sde-gdb* debugger how to connect to the MIPSsim simulator or EJTAG probe. If you will be using either of those then you'll need to enter:

a) A short name to identify this MDI device, e.g. "mipssim4", "sysnav", etc. Use the name "default" if you've only got one MDI device, or for the device which you expect to use most often.

b) A longer, more descriptive title for this device, e.g. "MIPSsim version 3.4.15" (don't enter the quote marks).

c) In the case of the MIPSsim software, the name of the directory or folder where you installed it (the same as the MIPSARCHROOT setting in the MIPSsim Guide) – if you have more than one version of the MIPSsim software installed then you can set up a separate fragment for each one, each with a unique name;

d) If you have an FS2 probe, then the script will search your path for the FS2 MDI library and create a configuration file for it automatically.

If you install MIPSsim software or probe drivers after installing SDE, then you'll have to perform this configuration step manually, as described in Section 14.1 "MDI Debugging".

7) To ensure that your new tools are immediately available to you, either close your shell window and reopen it, or run the commands displayed at the end of the `sdesetup` script, e.g.

```
$ . ./bin/sdeenv.sh              on bash, ksh, etc

% source ./bin/sdeenv.csh        on csh and tcsh
```

Now proceed to the next chapter to try out SDE on a simple example.

3.4 Multi-User Installation

If you want to install a single copy of the SDE toolchain to be shared by a group of programmers, simply follow the instructions above, but install the release into a well-known, shared directory, e.g. `/usr/local/sde6` or `/opt/sde6`.

You will then need to give each user their own copies of the `.../sde/kit` and `.../sde/examples` directories, so that they can build libraries and programs without interfering with each other. As long as the two directories remain at the same level (e.g. `~jones/sde6/sde/kit` and `~jones/sde6/sde/examples`) then the example makefiles will work correctly. The '`sdemklocal`' script handles this with a single command, for example:

```
$ ./bin/sdemklocal --destdir=~/jones/sde6
```

You can also maintain a single, shared kit run-time library tree, while building application programs in private directories. Simply point the **SDETOP** Makefile variable to the top of the shared SDE source tree, for example:

```
$ cd ~jones/sde6/sde/examples/hello
$ sde-make SDETOP=/usr/local/sde6/sde
```

Information for Upgraders

If you are upgrading from the SDE 5 series, then the new GNU toolchain may require some changes to your source code and/or Makefiles.

- *GCC −mcpu option* : The GCC compiler no longer supports the **−mcpu** flag, which has been replaced by **−mtune** and **−march** to independently control the pipeline scheduling, and the instruction set respectively. Usually just replacing **−mcpu=** by **−mtune=** in your Makefiles will work. This is handled automatically in the supplied example Makefiles.
- *The #cpu assertion* : In previous versions of SDE the **−mcpu** option also made the "`#cpu()`" assertion based on its argument, but this is no longer possible. It is now up to your Makefiles to explicitly make these assertions (e.g. using the '**−Acpu()**' flag), if you need them in your source code. The SDE Makefile system does this automatically based on the settings of the `CPU` and optional `CPUVARIANT` variables defined in your target's `sbd.mk` file.
- *Optimization levels* : Previous versions of SDE slightly redefined the meanings of GCC's optimization options to more closely match the original MIPS Corp compiler: so **−O3** enabled loop unrolling, and **−O4** enabled both loop unrolling and function inlining. We are now moving towards greater compatibility with standard GCC, so **−O3** enables function inlining, and loop unrolling requires the explicit **−funroll−loops** option.

  The following tables summarises the compiler and assembler command-line option changes which you might need to make to existing Makefiles:

Table 4-1 Modified compiler flags

SDE v5	SDE v6
−mcpu=xx	**−mtune=xx '−Acpu(xx)'**
−O3	**−O3 −funroll−loops −fno−inline**
−O4	**−O3 −funroll−loops**

  For more details and guidance see Section 12.2 "Optimization Options".

- *Legacy ISAs* : SDE no longer supports the legacy MIPS I, MIPS II, MIPS III and MIPS IV ISAs − only MIPS32 and onwards. You can instruct the compiler to generate code for the older ISAs, but we no longer test support for these ISAs, and you'll have to build your own libraries.
- *Linker −oformat option* : This ambiguous option (which could be confused with the catenation of **−o format**) was previously deprecated, but has now been totally removed from the linker. If it is still in your Makefiles, then it should be replaced by **−−oformat**.

  Furthermore the `elf32-littlemips` object format has been replaced by the `elf32-tradlittlemips` format, for compatibility with native Linux/MIPS toolchains. Similarly for `elf32-bigmips`. If you specify the object format explicitly in your Makefiles, then you will need to change the name accordingly. Better still remove the explicit object format, and the linker will then pick the correct format automatically − just use **-EL** or **-EB** to select the endianness.
- *Language standards* : The GCC compiler is somewhat more picky about enforcing language standards. In particular some GCC extensions have now been deprecated, which means that some programs which previously compiled without error or warning may need to be modified to be standards compliant. See `http://gcc.gnu.org/gcc-3.4/changes.html` for more information about changes in this version of the compiler.

  Issues which we've already tripped over are:

  - The cast-as-lvalue extension has been removed for C++ and deprecated for C and Objective-C. In particular, code like this:

```
int i;
(char) i = 5;
```

    or this:

```
char *p;
((int *) p)++;
```

is no longer accepted for C++ and will not be accepted for C in a future version.

- The undocumented extension that allowed C programs to have a label at the end of a compound statement, which has been deprecated since GCC 3.0, has been removed. So this:

```
switch (x)
{
default:
}
```

must be replaced by:

```
switch (x)
{
default:
    break;
}
```

- *Unused functions and variables* : If a module contains any *static* functions or variables which are not referenced within the same module, then the compiler will optimise them away altogether. This could be a problem if they are referenced implicitly, perhaps by virtue of their position within a linker script. Attach "`__attribute__((used))`" to such functions and variables to prevent them from being deleted; or as a temporary workaround compile with the **–fno–unit–at–a–time** option.
- *Function/variable order* : The new unit-at-a-time optimization means that functions and variables may now be output in apparently random order, rather than the order in which they appear in the source file. Try not to rely on the relative addresses of functions and variables. If you can't avoid this then use the **–fno–unit–at–a–time** option.
- *Strict Aliasing* : This version of GCC will now warn you by default about code which might break the strict aliasing rules that the compiler uses at higher optimization levels. See the description of the **–fstrict–aliasing** option in the [Gcc] reference manual for more details.
- *Assembler preprocessing* : Previous versions of SDE forced the C preprocessor to operate in "traditional" mode when processing assembler source files. This was done for compatibility with the old MIPS Corp. and SGI assemblers, but we no longer do this, in order to improve compatibility with standard GNU toolchains. The implications are:

  1) If you use symbol concatenation in your assembler source files or headers, then they must now use the ISO C "##" token pasting syntax, e.g.

```
#define MYSYM(x)        SYM_/**/x       // WRONG
#define MYSYM(x)        SYM_ ## x       // Correct
```

  2) The MIPS assembler convention of using '#' as an assembler comment leader now needs some care, because the modern ISO C language definitions allow a preprocessor directive to start in any column, not just the traditional column 1. So the following example may now be misinterpreted as containing two erroneous preprocessor directives:

```
label:
        # if this is the first word in a comment, also when you
        # include a line like this.
        addu    sp,-32
```

  3) You can specify **–traditional–cpp** to get the old SDE behaviour. The **–no–traditional–cpp** option has been removed, since this is now the default.

- *Mergeable constant data* : The compiler now emits constant strings and literal/immediate data (but not *const* variables) into the new `.rodata.str` and `.rodata.cst` sections, and the linker then merges identical constants within these sections from across your whole program into a single copy. You may see linker relocation errors if you use your own link script, and these new sections are not included within it. In such cases

you must add the extra `.rodata.*` wildcard input section to your linker script, just following the `.rodata` input section. For example:

```
.rodata {
        *(.rodata)
        *(.rodata.*)
}
```

Alternatively the new **–fno–merge–constants** compiler option can be used to disable the new mergeable sections altogether.

- *Insight GUI command*: The Insight GUI interface to GDB is now invoked using the new **sde–insight** command, and not **sde–gdb**. The latter now provides a command-line interface only.
- *DWARF-2 debug format*: The SDE v5 release used STABS as its default debug data format, but this is now deprecated and we have switched to the more powerful DWARF-2 format. If this is important to you, then you can specify **–gstabs** to select the old STABS debug format.
- *New 64-bit ABI*: Previous versions of SDE implemented a 64-bit calling convention (or ABI) which was an extension of the traditional, 32-bit "O32" ABI, and which provided some limited intercallability between 32- and 64-bit code. This requirement no longer seemed as important as achieving better 64-bit performance; avoiding non-standard and incompatible ABIs; and reducing major differences from "standard" GCC. So SDE 6 no longer supports the unusual "O32+" ABI, and selects the "N32" ABI when a 64-bit ISA is selected. The N32 ABI was chosen over N64 because it has a smaller memory footprint, and provides easier porting of existing 32-bit programs. See Section 12.6.1 "64-bit Calling Conventions".

Quick Start

If you are impatient to try out SDE, or want to confirm that your software installed OK, then follow these instructions to build and run one of the example programs using the GNU MIPS simulator (*sde−run*). If you have problems at any stage, support can be on hand; see Chapter 24 “Getting Support” for contact information.

In this example we’re going to use the **GSIM32L** target, which implies: a GNU simulator “target”; MIPS32® code; little-endian.

1) If you are running on Windows, then open a Cygwin shell window.

2) Change directory to the “hello world” example program:

```
$ cd .../sde/examples/hello
```

We’re not going to show you native Windows pathnames, though you can use them (with some caveats): see the notes on pathnames in Section 2.1.1 “File pathnames in Windows with Cygwin”.

3) Build the example (the upper/lower case distinction IS important):

```
$ sde-make SBD=GSIM32L
```

4) Run the program using the GNU simulator:

```
$ sde-run helloram
```

5) You can also run the program using the GNU debugger in command-line mode (same simulator):

```
$ sde-gdb helloram
(gdb) target sim
(gdb) load
(gdb) run
...
(gdb) quit
```

6) Try running the program using the *Insight* graphical interface to *gdb*:

   i) Start *gdb* with the command “**`sde-insight helloram`**”

   ii) The main Insight *Source Window* should open. If the *Console Window* doesn’t also appear, then click on the “console” icon in the source window’s toolbar. This allows you to see output messages from the program being debugged.

   iii) Click the “Run” icon (the running man) in the source window toolbar – the *Target Connection* dialog box will appear. Select “GNU Simulator” in the *Target* field of the dialog box, and click “OK”.

   iv) The program will be “downloaded” to the simulator, then run until it hits a breakpoint in `main()`.

   v) Click the “Continue” button (→{}) on the toolbar. The program will print “Hello World!” in the console window, and then stop at the next breakpoint, in the C library `exit()` function.

   vi) Select “Exit” from the source window’s “File” menu.

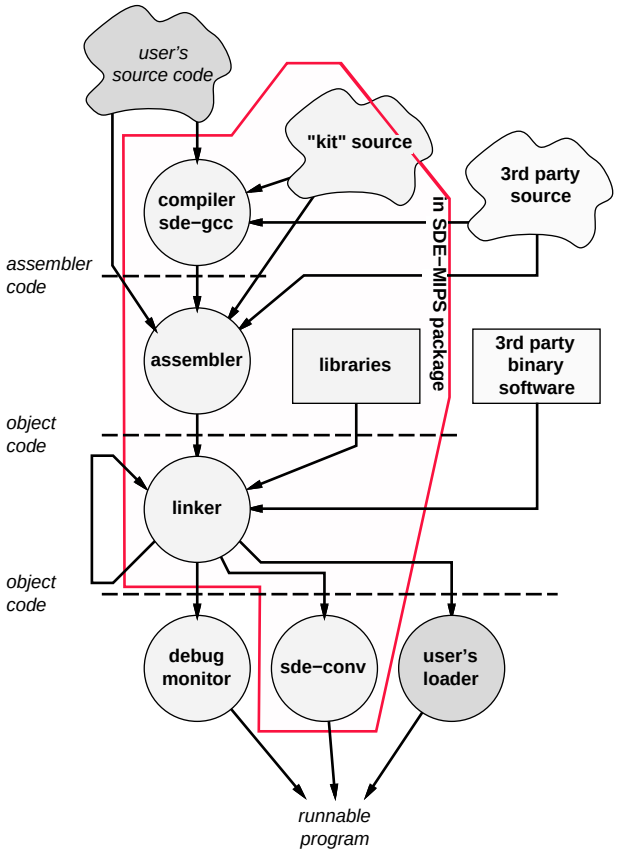
   See Chapter 13 “Insight Graphical Debugger” for more details. If you now want to try porting your own program to run on the GNU simulator, then see Chapter 10 “Porting an ISO / ANSI C Program”, which provides guidelines on porting ISO and POSIX standard C programs with SDE. If you want to try running example programs on real hardware, or on a more accurate software model such as the MIPSsim simulator, then see Chapter 8 “Target Specific Libraries” and/or Section 14.1.1 “MDI Debugging with the MIPSsim™ Simulator”.

Don’t forget that detailed manuals can be viewed with your web browser, see Chapter 7 “Online Documentation”.

Overview

This section provides a quick overview of the major components of SDE, particularly aimed at those for whom a command-line interface is not obviously a good idea.

Figure 6-1 Programs, libraries and source files in SDE



In Figure 6-1:

- The round objects are programs you run. You don't often need to know of the programs which really compile, assemble and link: they are generally orchestrated by the single "driver" program *sde−gcc*.
- The dark grey objects show the user-supplied files; intermediate grey are in SDE, and the light ones might be third-party software, if you use it.

Command lines, make and makefiles

An "integrated development environment" (IDE) like Microsoft's *Visual C*™ has become the standard development tool in the PC world. IDEs tie the basic compilation tools and libraries into a complex web of debuggers, editors, and other software.

By contrast, UNIX® command line tools were designed to be glued together using simple text files – shell scripts and "makefiles" (we'll say something about those just below). It's much simpler for us to supply and maintain individual tools which build on all the wonderful free software that's available for the MIPS architecture.

Program Editor

Since we're not supplying an integrated environment, you need to bring your own program editor. There are lots of fairly good ones about; it's worth rooting around on the web, since you probably spend a lot of your life editing and the obvious tools available on every PC are pretty basic. The author (and most of the SDE team) use and warmly recommend XEmacs or GNU Emacs, which take a while to learn but are very powerful. Nowadays one or both flavors of Emacs are often a standard package in modern UNIX-like distributions, including Cygwin – alternatively visit `http://www.xemacs.org`. But we understand that this is not so much an editor, more a way of life; editors are a matter of personal preference.

Your editor should allow you to fire off compilations without quitting the editor, catch any compiler errors and interpret them to automatically locate you in the file where the error was detected. Many decent editors can understand GNU C's error formats well enough to do this, though you might have to customise them to do this.

Make

Once you've got your editor, the job of SDE (whether IDE or command-line based) is to take a bunch of source files, libraries and the like which are to make up one or more useful program(s), and to run appropriate compilation, assembly and link operations until you get a runnable program.

When you're building applications ''natively'' to run on your host operating system most of the details are hidden by the compiler's OS defaults – most of the time. The less of an operating system you have, though, the more complicated the building process becomes.

In SDE the build job is directed by the *sde-make* program which finds out what to do to build a particular set of files from a plain text file which you've prepared – the *Makefile*. All the examples in SDE come with simple, reusable ''Makefiles'' ready to run.

For all the grisly detail see the GNU manual [Make], but here's a comforting four-paragraph guide:

In simple cases, where the source and target files all live in one directory, *make* will by default take its instructions from a file called `Makefile`.

Inside the makefile, you'll find entries which look a bit like this:

```
target: depend1 depend2 ...
        do-this
        do-that
```

(That's a tab character at the beginning of the action lines, not just spaces).

When asked to '`make target`', this will check to see whether the file target (if it is a file) is older (earlier write time) than any of the files `depend1` etc. If one of those files has been changed, it will run the commands *do-this*, *do-that* in sequence, just as if those command lines had been typed at the shell prompt.

If you don't specify a target for *make*, the default is the first target in the makefile; it's conventional to lay out the file so you can build the most obvious target in your ''project'' by just typing *make*.

Of course, over the years *make* has grown lots of other facilities, all of which seemed to be a good idea at the time, so a modern makefile is fairly scary – as is the GNU manual [Make]. Some important extra features include:

- *Wildcards in targets*: targets can be specified with wildcard names, like `*.c`, specifying the default action for files which look like that (you can override these with a specific entry).
- *Variables*: the ugly syntax ''`$(CFLAGS)`'' substitutes the (string) value of a variable `CFLAGS`, which may be set earlier in the makefile or inherited as an *environment variable*.
- *Included sub-makefiles*: lines starting `include ...` have the effect of calling in another makefile – just like a C `#include`, the lines of the file are treated just as if they were part of the original makefile. SDE uses the facility extensively, using nested makefiles to share information up and down its file hierarchy.

The golden rule of ''make'': *NEVER* write your own makefile (at least, not until you're experienced enough to understand why we said that). Instead, copy something vaguely like what you're trying to do and hack it into shape. That way, the bits you don't understand will just quietly carry across.

Oh, and don't put spaces or other non-alphanumeric characters in your file names; *make* will hate it.

C Compiler

This is our version of the Free Software Foundation's GNU C Compiler (called *sde-gcc*[11]). This incorporates superb optimization for the MIPS architecture, and also benefits from many of our own bug fixes, enhancements and optimizations in order to deliver the best possible results on MIPS-Based CPUs.

In practice, the C compiler, C++ compiler, assembler and linker are all usually invoked as *sde−gcc*, which (by default) figures out what to do with a file based on the filename extension.

C++ Compiler

We include the GNU C++ compiler (*sde-g++*). The *sde-g++* compiler supports modern C++ features, and benefits from all MIPS Technologies' enhancements and optimizations to the common back-end of GCC. However use of C++ exceptions and/or run-time type identification incur a significant size overhead. If these features are not required by your code, then they can be switched off individually using the **−fno−exceptions** and **−fno−rtti** options, respectively.

MIPS® Assembler

SDE's version of the GNU assembler (*sde−as*[12]) is, as far as is possible, source code compatible with the "standard" MIPS assembler syntax, including the modern MIPS32® and MIPS64® instruction sets and their "Release 2" variants, together with the historical MIPS I™ through MIPS V™ ISAs, and standard extensions like the MIPS16®, MIPS16e™, SmartMIPS®, MIPS-3D®, MDMX™, MIPS MT and MIPS DSP Application Specific Extenstions (ASEs).

Binary Utilities

The GNU binary utilities support a version of the ELF object code format. Our ELF is pretty compatible with other MIPS tools; ELF is probably the most widely used family of object codes for 32-bit CPUs[13]. The tools are described in detail in the GNU manual [Binutils] and include:

- *sde−ld* : the link editor/locater or linker − usually run automatically by *sde−gcc* rather than directly by name. It supports sophisticated link script files for building complex load images − see [Ld]. Such features are never without cost; if your system can use simple program images, your project will be blessed.
- *sde−size* : prints the size of the various sections in an object file.
- *sde−nm* : prints the names held in an object file's symbol table, sorted by address or by name.
- *sde−strip* : removes an object file's symbol table, to save on disk space.
- *sde-ar* : an object code archiver/librarian.
- *sde−objdump* : prints out parts of object files for inspection, including disassembly of code sections.
- *sde−strings* : displays any readable ASCII text strings in an object file.
- *sde−objcopy* : copies object files, optionally converting object formats, and including or excluding named sections.
- *sde−gprof* : profiling report generator, with its own manual [Gprof].
- *sde−readelf* : reports the low-level structure of an ELF object file (use *sde−objdump* to read the contents).

[11] All the GNU tools are named like this; it avoids name clashes with other versions of GNU CC which may be installed on your system. Previous versions of SDE also included tools without the *sde−* prefix, but these are no longer provided.

[12] It would be more consistent to call it *sde−gas*; the reasons for not doing so are historical.

[13] Don't assume that this means that software written for some other ELF dialect will port easily to the MIPS version. ELF is more a family of standards than a standard.

ECOFF compatibility

Most of the binary utilities such as *sde−ld* and *sde−objcopy* offer some support for object files and libraries in the ‘‘ECOFF’’ format produced by the 1980s MIPS Computer Systems native compiler.

Download Tools

To download the executable binary files produced by the linker to an evaluation platform or PROM programmer requires additional conversion and communication tools.

- *sde−conv* : converts a binary object file into a number of formats, including Motorola S-record, MIPS flash download, IDT/sim binary, LSI PMON fast format, and Stag (prom programmer) binary.

Libraries and Header Files

C is nothing without its libraries; SDE has the standard C library and math library supplied pre-compiled for a range of different MIPS ISA options; the version you need is picked automatically according to the flags you give the compiler – see Section 11.3 ‘‘Multilibs’’. Customers who purchase the MIPS® Software Toolkit also receive the full library source code – see Section 11.4 ‘‘Library Source Code’’.

The C libraries and associated header files follow the ISO C Standard (ISO 9899:1990[1992]), also known as ISO C90, and formerly the ANSI X3J11 committee’s standard for the C programming language (called the ANSI C standard in this manual). This has been validated using the Plum Hall Validation Suite. Some extensions from ISO C99 are also implemented. The low-level run-time i/o system is modelled on the POSIX.1 API.

CPUs without floating point hardware can take advantage of our IEEE−754 compliant floating point emulator, provided seperately in its own library for easy re-use.

In addition there are machine-specific header files covering a variety of MIPS architecture processors, and associated support chips.

Thread Safety

The C library and board-support code can be fully reentrant and thread-safe. A plug-in TSP (Thread Support Package) for SDE can provide integration between the SDE libraries and build tools, with a third-party real-time microkernel. Contact your MIPS Technologies representative for more information, or to order.

Embedded System Kit

Our valuable collection of low-level functions for handling reset-time initialization, and run-time management of caches, MMU, exceptions, interrupts and floating point coprocessor (including a trap-based emulator). It also provides a POSIX-like run-time i/o system.

MIPS® Software Toolkit customers (those on support) will get full source code of this kit; SDE *lite* users will find that many modules are provided as pre-compiled libraries, with the filename extension `.lib`.

Micromon

Only for MTK customers with source code. A tiny, RAM-less PROM monitor built on top of the low-level board initialization and console i/o code. It runs out of ROM, using only registers and a UART, and allows you to ‘‘peek’’ and ‘‘poke’’ memory and device registers using a simple reverse-polish command language. We find this to be be a useful tool when bringing up a new board design or system controller.

Example Programs

The collection of example programs provided with SDE:

1. Allows you to check out your installation, your hardware configuration, your host/hardware connection and other critical support functions. You don’t want to be debugging software until you know these things are right.
2. Provides example makefiles which you can copy and adapt to the programs you want to build.
3. In particular, provides examples of how to build and run benchmark programs.

4. Allow you to explore some more complex CPU-specific areas – interrupts, exceptions and so on.

Source Level Debugger

The GNU debugger (*sde−gdb*) provides sophisticated source and machine level debugging. The debugger has an optional graphical user interface known as“Insight”.

The debugger runs on your development host and communicates with the target – which can be real or simulated hardware, anything which runs MIPS instructions. For a real target board *sde−gdb* can either:

- connect to a monitor program on the target via a serial line, or over a TCP/IP network (either via a terminal concentrator, or directly to monitors with a TCP stack); OR
- use an on-chip debug unit if available, so long as the probe device and its host software provide an MDI or “gdb remote” interface; OR

GNU MIPS® CPU Simulator

A software simulator for MIPS architecture processors (*sde-run*) allows standalone programs to be debugged before the availability of working target hardware. It’s based on a GNU program. It can be used to run all of the supplied example programs.

Note that this is *only* a CPU emulator: to find a way of simulating your larger system you must look elsewhere.

The simulator is most often used from within *sde−gdb* – which it’s built into – to allow source level debug of simulated code.

MIPSsim™ Simulator

MIPS Technologies provide this much more comprehensive and accurate core simulator as a standard component of the MIPS® Software Toolkit package.

The *sde−gdb* debugger connects to the MIPSsim software using the MDI interface.

Online Documentation

As described in the very next chapter.

Online Documentation

As part of our distribution we have included both browsable (HTML) and printable (PDF) manuals for all of the major GNU components used in SDE.

Browsable HTML pages

The HTML versions of the GNU manuals are derived from the same text as the printable manuals, but you have the additional ability to navigate around the manuals using your favourite Web browser. On a UNIX system point your browser at the URL `file://.../html/index.html` (where as usual "`...`" is where you installed the software). On Windows use `file:\\\c:\cygwin\...\html\index.html` , assuming you installed Cygwin on drive C. You'll probably want to add that URL to your brower's bookmarks or "favorites" folder.

Printable manuals

Printable PDF versions of all SDE manuals are included in the distribution. There are links to them from the HTML pages, or you can locate them manually in `.../doc/`.

You should probably print a copy of the Programmer's Guide, which you are reading now. Many of you will find that you can make extensive use of the tools just by starting from our examples, and answering the occasional detailed question by looking at the GNU manuals. But the GCC manual may be worth a thorough read by any MIPS developer who really wants to get the best performance and maximise portability.

Target Specific Libraries

SDE's run-time system provides an identical software interface across a range of different evaluation boards and software simulators, known here as "targets". The run-time system is provided as full source code for MTK customers, but as pre-compiled object files for most other users. Under the control of a per-target configuration file it is built into a set of libraries specific to the chosen target. Much of the run-time code is generic and will work on any MIPS-Based target, but drivers specific to a range of MIPS Technologies boards and simulators are included. For MTK customers it is straight-forward to add a new target, as described in Chapter 22 "Retargetting the Toolkit". The supported target configurations are listed in Table 8-1, below. The columns are as follows:

- *Platform* : the evaluation board or software simulator.
- *CPU* : the supported CPU types.
- *Base ISA* : the base intruction set architecture. You can add variants like the MIPS16 ASE and the Release 2 extensions to this, see Section 8.1 "Building for ISA and CPU Variants".
- *FPU Type* : the floating point hardware model. "None" implies software floating point; "64-bit" implies a 64-bit h/w FPU with the CPU's *Status.FR* bit set; and "32-bit" implies either a 32-bit FPU, or a 64-bit FPU with the FR bit clear. See Section 12.5 "Software Floating Point" for more information.
- *Endian* : the CPU endianness. For a hardware target this must match the board's switch settings.
- *Connection* : how the *sde−gdb* debugger communicates with the target – "YAMON" implies a serial port connection to the YAMON™ monitor; "MDI+EJTAG" is an EJTAG probe with MDI debugger interface; "MTSPMON" refers to the Linux AP/RP *pseudo-monitor*.
- *SBD* : the "System Board Description", an identifier which describes this target to the SDE makefile system.

Table 8-1 Supported target boards and simulators

Platform	CPU(s)	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPS Atlas™	4Kc®, 4Km®, 4Kp®	MIPS32	None	BE	YAMON	ATLASLV4B
				LE		ATLASLV4L
MIPS SEAD-2™	4Kc, 4Km, 4Kp, 4KEc, 4KEm, 4KEp, 4KSc, 4KSd, M4K	MIPS32	None	LE	YAMON	SEAD32L
				BE		SEAD32B
				LE	MDI+EJTAG	SEAD32LJ
				BE		SEAD32BJ
	5Kf, 20Kc, 25Kf	MIPS32	32-bit	LE	YAMON	SEAD32FL
				BE		SEAD32FB
				LE	MDI+ EJTAG	SEAD32FLJ
				BE		SEAD32FBJ
	24Kf, 24KEf, 34Kf	MIPS32 Release 2	64-bit	LE	YAMON	SEAD32F64L
				BE		SEAD32F64B
				LE	MDI+EJTAG	SEAD32F64LJ
				BE		SEAD32F64BJ
	5Kc	MIPS64	None	LE	YAMON	SEAD64L
				BE		SEAD64B
				LE	MDI+EJTAG	SEAD64LJ
				BE		SEAD64BJ
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE	YAMON	SEAD64FL
				BE		SEAD64FB
				LE	MDI+EJTAG	SEAD64FLJ
				BE		SEAD64FBJ

Platform	CPU(s)	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPS Malta™	M4K™	MIPS16e	None	LE	YAMON	MALTAM4KL
				BE		MALTAM4KB
				LE	MDI+EJTAG	MALTAM4KLJ
				BE		MALTAM4KBJ
	4Kc, 4Km, 4Kp, 4KEc®, 4KEm™, 4KEp™, 4KSc™, 4KSd™, M4K, 5Kc™, 24Kc™, 24KEc™, 34Kc™, 74Kc™	MIPS16e	None	LE	YAMON	MALTA16L
				BE		MALTA16B
				LE	MDI+EJTAG	MALTA16LJ
				BE		MALTA16BJ
	4Kc, 4Km, 4Kp, 4KSc™, 5Kc™,	MIPS32	None	LE	YAMON	MALTA32L
				BE		MALTA32B
				LE	MDI+EJTAG	MALTA32LJ
				BE		MALTA32BJ
	4KEc®, 4KEm™, 4KEp™, 4KSd™, M4K, 24Kc™, 24KEc™, 34Kc™, 74Kc™	MIPS32 Release 2	None	LE	YAMON	MALTA32R2L
				BE		MALTA32R2B
				LE	MDI+EJTAG	MALTA32R2LJ
				BE		MALTA32R2BJ
	5Kf®, 20Kc™, 25Kf™	MIPS32	32-bit	LE	YAMON	MALTA32FL
				BE		MALTA32FB
				LE	MDI+EJTAG	MALTA32FLJ
				BE		MALTA32FBJ
	24Kf™, 24KEf™, 34Kf™, 74Kf™	MIPS16e	64-bit	LE	YAMON	MALTA16FL
				BE		MALTA16FB
				LE	MDI+EJTAG	MALTA16FLJ
				BE		MALTA16FBJ
		MIPS32 Release 2	64-bit	LE	YAMON	MALTA32R2FL
				BE		MALTA32R2FB
				LE	MDI+EJTAG	MALTA32R2FLJ
				BE		MALTA32R2FBJ
	34Kc, 34Kf	MIPS32 R2 + MT ASE	None	LE	MTSPMON	MALTA32LSP
				BE		MALTA32BSP
				LE	YAMON	MALTA32MTL
				BE		MALTA32MTB
				LE	MDI+EJTAG	MALTA32MTLJ
				BE		MALTA32MTBJ
	34Kf		64-bit	LE	YAMON	MALTA32MTFL
				BE		MALTA32MTFB
				LE	MDI+EJTAG	MALTA32MTFLJ
				BE		MALTA32MTFBJ
	5Kc	MIPS64	None	LE	YAMON	MALTA64L
				BE		MALTA64B
				LE	MDI+EJTAG	MALTA64LJ
				BE		MALTA64BJ
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE	YAMON	MALTA64FL
				BE		MALTA64FB
				LE	MDI+EJTAG	MALTA64FLJ
				BE		MALTA64FBJ

Platform	CPU(s)	Base ISA	FPU Type	Endian	Debug Conn	SBD
MIPSsim	M4K	MIPS16e	None	LE	MDI	MSIMM4KL
				BE		MSIMM4KB
	4Kc, 4Km, 4Kp, 4KEc, 4KEm, 4KEp, 4KSc, 4KSd, M4K, 5Kc, 24Kc, 24KEc, 34Kc, 74Kc	MIPS16e	None	LE	MDI	MSIM16L
				BE		MSIM16B
	4Kc, 4Km, 4Kp, 4KSc, 5Kc	MIPS32	None	LE	MDI	MSIM32L
				BE		MSIM32B
	4KEc, 4KEm, 4KEp, 4KSd, M4K, 24Kc, 24KEc, 34Kc, 74Kc	MIPS32 Release 2	None	LE	MDI	MSIM32R2L
				BE		MSIM32R2B
	5Kf, 20Kc, 25Kf	MIPS32	32-bit	LE	MDI	MSIM32FL
				BE		MSIM32FB
	24Kf, 24KEf, 34Kf, 74Kf	MIPS16e	64-bit	LE	MDI	MSIM16FL
				BE		MSIM16FB
		MIPS32 Release 2	64-bit	LE	MDI	MSIM32R2FL
				BE		MSIM32R2FB
	34Kc	MIPS32 R2 + MT ASE	None	LE	MDI	MSIM32MTL
				BE		MSIM32MTB
	34Kf		64-bit	LE	MDI	MSIM32MTFL
				BE		MSIM32MTFB
	5Kc	MIPS64	None	LE	MDI	MSIM64L
				BE		MSIM64B
	5Kf, 20Kc, 25Kf	MIPS64	64-bit	LE	MDI	MSIM64FL
				BE		MSIM64FB
GNU simulator	any	MIPS32	32-bit	LE	builtin	GSIM32L
				BE		GSIM32B
	any	MIPS16e	32-bit	LE	builtin	GSIM16EL
				BE		GSIM16EB
	any	MIPS64	64-bit	LE	builtin	GSIM64L
				BE		GSIM64B

The **SBD** column gives the short-form name of the board. This name identifies the sub-directory of `.../sde/kit` which contains the configuration files and possibly driver source code for this target. So, for example, the directory `.../sde/kit/MALTA32L` holds the target-specific information and code for MIPS Technologies' Malta board, with a MIPS32 CPU, without h/w floating point, little-endian, debugging via a serial connection to the YAMON monitor.

To build the run-time library for one of the above targets, you simply go to its directory and run *sde−make*:

```
$ cd .../kit/MALTA32L
$ sde-make
```

Having successfully built the library, you can then build any or all of the example programs. When building an example the first time, you need to specify the value of **SBD** on the *sde−make* command line:

```
$ cd .../examples/hello
$ sde-make SBD=MALTA32L
```

This creates a file named MALTA32L.sbd in the working directory which records **SBD** and **SDETOP**; further make makes will pick them up as default values. When you upgrade to a newer version of SDE, remove all generated files with:

```
$ sde-make clobber
```

Note: Specifying a different **SBD** value will cause the example *makefiles* to delete all object files, and rebuild the program.

8.1 Building for ISA and CPU Variants

Due to the large range of processor cores and different ISAs and ASEs which are available on MIPS Technologies eval boards and simulators, the run-time libraries for the Malta and SEAD-2 evaluation boards and the MIPSsim simulator are configured for just a small number of base-level ISAs – see Table 8-1 "Supported target boards and simulators" above. If you want to build an application or benchmark which exploits a particular extended ISA or ASE, such as the MIPS32 Release 2 ISA, or the SmartMIPS and MIPS16e ASE, then this is easily done when building your application by using the Makefiles' `APPISA` variable (see Section 9.2 "Example Makefiles"). Just pick the value of `SBD` which most closely matches your target "board" and CPU configuration, and then specify the extended ISA as follows:

```
$ cd .../sde/examples/ex5
$ sde-make SBD=MSIM32L APPISA=-mips32r2
$ sde-make SBD=MSIM32L APPISA="-mips32 -mips16"
$ sde-make SBD=MSIM32R2L APPISA="-mips32r2 -msmartmips"
$ sde-make SBD=MSIM32R2L APPISA="-mips32r2 -mdsp"
```

See Section 12.1 "Architectural Flags" for a full list of the ISA options.

Similarly you can optimize the application for a specific CPU type using the APPCPU variable, for example:

```
$ cd .../sde/examples/dhrystone
$ sde-make SBD=MSIM32R2L APPCPU=74kc
```

See Table 12-1 "List of –mtune= names" for a full list of supported CPU types.

Example Programs

The `.../sde/examples` directory contains several small programs which demonstrate the use of SDE. They are each held in individual sub-directories, listed below, and they can all be built to execute in RAM under the control of a board's PROM monitor, or via an EJTAG probe, or (on some targets) blown into ROM, or run by a simulator.

All of the examples are built under the control of a common include file `.../sde/examples/make.mk`, which uses the board-specific parameters selected by the **`SBD`** variable to compile and link each program with the correct compiler flags and libraries.

We suggest that you first try building the examples and running them with the GNU simulator, to see how they behave. This procedure is fully described in Chapter 5 "Quick Start".

When you are happy with this you can build the board-specific library for your target as documented in Chapter 8 "Target Specific Libraries", and then rebuild the examples. Instructions on how to download and run programs on the supported boards can be found in Chapter 14 "Debugging with GDB" and Chapter 17 "Manual Downloading".

The remainder of this chapter describes the purpose of each example program.

9.1 Individual Examples

9.1.1 Hello World!

The program in `.../sde/examples/hello/hello.c` is simply everyone's first C program – just to get you started!

9.1.2 TLB Exception Handling (tlbxcpt)

The example in `.../sde/examples/tlbxcpt` introduces SDE's "C" interface to low-level CPU exceptions. These are called *xcptions*, and are described in Section 20.2.1 "C-level Exceptions". This program randomly accesses memory via the mapped KUSEG and KSEG2 regions (MIPS architecture magic words, read [Sweet99] if you don't know what they mean). On catching the resulting "TLB Miss" exceptions it updates the TLB and returns to the faulting instruction. On completion it displays the number of TLB misses.

Note that some MIPS-Based CPUs don't have a TLB, and they will not be able to run this example.

9.1.3 Command Line Monitor (minimon)

This example provides a very simple command line monitor program, which is actually quite useful for peeking and poking devices on a new target, and can form the basis of useful command-line test harnesses. Type '`help`' at it for a list of commands.

One thing to note in this program is its use of POSIX *signal*-handling to catch address errors, and to test SDE's interval timing functions, see Section 19.1 "POSIX API Environment". In fact the program was written and tested on a UNIX system before being ported to SDE.

This example might also be a good one with which to try out the *sde-gdb* debugger. If you reference an invalid address with the *put* or *get* commands (e.g. "**`g 1`**" will cause an address exception), then the debugger will be entered, allowing you to examine the cause of the exception. See Chapter 14 "Debugging with GDB" for more information on this procedure.

Another useful piece of example code provided within this program is an ELF object file loader, which can load an ELF executable from a supported file-like device into memory – for example a flash ROM. See the `com_boot` function.

The ELF file loader is also capable of loading, relocating and then invoking a self-contained position-independent dynamic shared object (DSO) file. Self-contained means that the shared object must contain no undefined external references – the loader isn't yet smart enough to resolve symbols. You can try this out on a simulator target, as follows:

1) Build and run the minimon example for a simulator target, for example:

```
$ sde-make SBD=MSIM32L
```

2) Build the example DSO as follows:

```
$ sde-make SBD=MSIM32L dso
```

3) Load and run the minimon example on a simulator:

```
$ sde-gdb miniram
(gdb) target mdi 15:1
(gdb) load
(gdb) run
minimon> boot dso
```

9.1.4 Floating Point Test (paranoia)

The source file `.../sde/examples/paranoia/paranoia.c` is a public domain program, originally written by one of the creators of the IEEE−754 floating point standard. It is used to test many aspects of the standard: from the basic arithmetic, to the niggly rounding modes, overflow, underflow etc. We use it to test our software floating point emulation. You can use it to check that the floating point infrastructure of SDE is correctly installed and configured for your target.

9.1.5 Dhrystone Benchmark

The well known *dhrystone* benchmark (version 2.1) is in `.../sde/examples/dhrystone/dhry.c`. It serves as an example of how to port a simple integer-only benchmark. It only required configuration to use the ISO / ANSI `clock()` function for its timing, and a minor change to disable it from attempting to write its results to a disk file.

The *makefile* for this example switches on high optimization (**−O3**).

Note that when using the MIPSsim simulator the elapsed time for benchmarks is calculated from the simulator's cycle count, and then assuming that the simulated CPU is running at only 100 kHz (with a 300MHz PC that will actually be close to real time, since the simulator runs at about 3000 instructions to 1) − you'll then have to scale the elapsed time to get a correct result for the expected target CPU frequency (e.g. for a 250MHz target divide the elapsed time by 2500, or multiply the benchmark result by 2500).

The GNU simulator can be used to debug benchmark programs like *dhrystone*, but it is an "instruction" simulator only. It makes no attempt to be cycle accurate, and does not simulate hardware timers or clocks, so programs will display a zero elapsed time. To get representative timings of simulated benchmark code you must use MIPSsim.

9.1.6 Whetstone Benchmark

The double-precision *whetstone* benchmark is in `.../sde/examples/whetstone/whetd.c`. It is an example of how to port a floating point benchmark. The only change was to make it use the ISO / ANSI `clock()` function to do its timing. It is built with high optimization (**−O3 −ffast−math**).

For more information on the use of floating point, see Section 9.2 "Example Makefiles" and Section 12.5 "Software Floating Point".

9.1.7 Linpack Benchmark

Another well-known floating point benchmark is in directory `.../sde/examples/linpack`.

9.1.8 C++ Demo

This example builds a small C++ program: `.../sde/examples/cxxtest/tstring.cc` is a string handling test program from the GNU *libstdc++* library. If you would like to contribute a more interesting self-contained example, then please let us know!

9.1.9 Kit Test

This example `.../sde/examples/kittest/hello.c` is another "Hello World" program, but one which has a real purpose: it contains code that performs a simple confidence test of your target's memory system, serial port, "system interface" code and C library i/o functions.

If you are retargetting SDE to a new board, then you must make sure that this program runs before any other – basic console output must work before you stand a chance with anything more complex. In particular don't try to use the SDE remote debug stub with this example, since the debug facility uses precisely the code that you are testing here. So if your new target-specific code doesn't work well enough to run this program and talk to a serial port, then you'll need to debug it with an EJTAG probe, a logic analyser, or a pre-existing PROM monitor.

9.1.10 Flash Memory Test

The example program in `.../sde/examples/flash/flashtest.c` tests a board's Flash memory system (programming and erasing) and demonstrates use of the facilities described in Section 19.1.4 "Flash Memory Device (/dev/flash)".

Note that the Makefile defines `FEATURES=flashdev` to include the Flash device driver in the build, see Section 9.2 "Example Makefiles" for details.

9.1.11 PCI Bus Demo

The example program in `.../sde/examples/pci/pcitest.c` which demonstrates how to setup, probe and access a board's PCI bus and PCI devices using the facilities described in Section 19.2 "PCI Bus Support".

The example enumerates all devices on the bus and displays their configuration space registers symbolically. If the device has a boot ROM (and the target is running little-endian), then the ROM is accessed and its headers are decoded.

9.1.12 Decompressing Boot Loader

The example program in `.../sde/examples/zload/zload.c` is a small decompressing boot loader which could be used to load into RAM an application which is too big to fit into ROM. It also demonstrates use of the front-panel display device described in Section 19.1.5 "Alpha Display (/dev/panel)".

Note that the Makefile defines `FEATURES=paneldev` to include the front-panel display driver in the build, see Section 9.2 "Example Makefiles" for details.

The Makefile will automatically compile and link a tiny program `exec.c` into an ELF executable file and compress it. If you then run this example program on a simulator, or other target which support virtual host i/o, then it will read the compressed program, decompress it, load it into memory, and call it.

9.1.13 Linux AP/RP Communication

The example program in `.../sde/examples/rtlx/rtlx.c` demonstrates the low-level communication mechanism between a program running in the "Real-time Processor" of a multi-VPE MIPS CPU, communicating with a Linux kernel device driver running on the "Application Processor". It uses the character device files described in Section 19.1.3 "Linux AP/RP Communication (/dev/lx#)", which will only work in conjunction with one of the target board kits which support the `mtspmon` interface, namely: `MALTA32LSP` or `MALTA32BSP`.

For information about debugging AP/RP programs, see Section 14.2.3 "Debugging AP/RP Applications".

9.1.14 Interrupt Example

The example program in `.../sde/examples/spxcpt/spxcpt.c` demonstrates how to install an interrupt handler on the Malta platform. It installs an interrupt handler which updates the LED display every 0.1 seconds.

9.2 Example Makefiles

Each example sub-directory contains the source of the program and a *makefile*. Each *makefile* defines a few variables and then includes the common file `.../sde/examples/make.mk`. This rather complicated makefile uses the board-specific parameters defined in the kit directory `.../sde/kit/$SBD/sbd.mk` to build each program with the correct combination of compiler flags and libraries to match the CPU type, endianness, floating point hardware, etc. on the selected target board.

The default action of `make.mk` is to build three versions of your program: downloadable using ROM monitor, downloadable but with its own I/O routines, and rommable. So for example the *dhrystone* benchmark makefile, which defines '`PROG=dhry`', will generate (along with a number of intermediate files) files named like this:

Table 9-1 Example Makefile output files

Filename	*Purpose*
`dhryram`	An executable file linked for downloading into RAM, and running with the board's PROM monitor. Some monitors can load this file directly over Ethernet.
`dhryram.dl`	The above executable, converted into a format suitable to transfer over a serial link to the board. The "`.dl`" is one of the formats supported by the *sde−conv* program.
`dhrysa`	A standalone executable file, linked for a RAM address, but once downloaded and started is independent of the PROM monitor (i.e. it includes its own UART drivers, etc).
`dhrysa.dl`	The standalone executable converted into download records, suitable for your PROM monitor.
`dhryrom`	A rommable executable file – it may relocate itself to RAM if required for debugging, or if requested by the **`LAYOUT`** variable (see below).
`dhryrom.s3`	The rommable executable, converted into Motorola S-records ready to transfer to your PROM programmer.
`dhryrel`	For AP/RP configurations, a relocatable version of your program which can be loaded at run-time by the operating system.

The operation of `make.mk` can be further controlled by setting additional variables, in one of the following ways:

1) Specify the variables on the command line, e.g.

```
$ sde-make SBD=MALTA32L APPISA="-mips32 -mips16e"
$ sde-make SBD=MSIM32R2L APPCPU=4ksd APPISA="-mips32r2 -msmartmips"
$ sde-make SBD=MSIM32R2L APPCPU=24kec APPISA="-mips32r2 -mdsp"
```

   Note the use of quotes around the command-line values which contains spaces.

2) Edit one of the example *makefiles* only, so that just that one program is affected, and add lines which define the relevant variables, e.g.:

```
SBD=MALTA32R2FL
APPCPU=4ksd
APPISA=-mips32r2 -msmartmips
```

   Note how, in a Makefile, values with spaces do not require quotes.

3) Add the same lines to `.../sde/examples/make.mk` so that they will apply globally to all *makefiles* which use it.

4) Set them as environment variables. For example with Bourne shell or similar:

```
$ SBD=MALTA32R2FL; export SBD
$ APPCPU=4ksd; export APPCPU
$ APPISA="-mips32r2 -msmartmips"; export APPISA
```

   or with C shell:

```
% setenv SBD MALTA32R2L
% setenv APPCPU 4ksd
% setenv APPISA "-mips32r2 -msmartmips"
```

You can have the environment variables set every time you use the software by editing a startup script; see Section 3.2 “Environment Variable Setup” for advice.

The list of variables that you may want to change is as follows:

Table 9-2 User-changeable “Make” variables for program building

Variable Name	Default Value	Permissible Values	Description
ALL	`rom ram sa`	*any*	The default list of files to build.
APPCPU	`$(CPU)`		Override the default CPU type.
APPISA	`$(ISA)`		Override the default ISA.
ASFLAGS	`$(CFLAGS)`	*any*	Assembler flags.
CFLAGS	`-O2 -g`	*any*	C compiler flags.
CPPFLAGS		*any*	C pre-processor flags (e.g. `-D`, `-U`, `-A`, etc) to use when compiling the application source code.
CRT0FLAGS		*any*	Additional C pre-processor flags for customizing the `crt0.o` startup module:
		`-DMINKIT`	Don’t de-initialise full POSIX run-time library, see Section 11.1.3 “Minimal C library”.
		`-DSMALLXCPT`	Don’t initialise early exception handling, see Section 11.1.3 “Minimal C library”.
		`-DNOCTORDTOR`	No support for constructors and destructors. see Section 11.1.3 “Minimal C library”.
		`-DNOFEATUREINIT`	No initialization of library features. see Section 11.1.3 “Minimal C library”.
CXXFLAGS	`-O2 -g`	*any*	C++ compiler flags.

Variable Name	Default Value	Permissible Values	Description
FEATURES	A list of run-time "features", separated by spaces, which you want to include or exclude from your application. Wild-cards can be specified using the '%' character, e.g. 'FEATURES=pci%'. To request an feature optionally, prepend a '/' character, e.g. 'FEATURES=/paneldev'. The currently supported feature list is:		
		all	Include all optional run-time features supported on this board. To then explicitly exclude some features, append the feature names preceded by '–', e.g. 'FEATURES=all -pci%'.
		flashdev	The /dev/flash interface, see Section 19.1.4 "Flash Memory Device (/dev/flash)"
		paneldev	The /dev/panel interface, see Section 19.1.5 "Alpha Display (/dev/panel)"
		pci	The PCI bus scannning and initialization code. Thie will be included automatically if any of the PCI support functions are called by your code.
		pcilookup	Lookup table to translate known PCI vendor and device IDs to readable names. This table currently occupies 40KB and will only grow!
		unaligned	Install an unaligned address exception handler to fix up occasional unaligned accesses. But don't use this in production code, it will be very slow!
		xcptstackinfo	Stack backtrace on fatal exception (default in ROM code with remote debugging enabled)
FLOAT	no	no	floating point is not used.
		yes	Basic floating point support required.
		ieee	Full IEEE–754 conformance (NB this may increase program size significantly).
LAYOUT	rom	rom	Copy only initialised data to RAM; run code from ROM.
		romcopy, ram	Copy both code and initialised data from ROM to RAM for better performance, or to set software breakpoints. This is the default if RDEBUG=imm is specified.
LDFLAGS		*any*	Additional linker flags.
LDSCRIPT		*any*	Custom linker script which overrides the standard one.
LDLIBS		*any*	Additional local libraries on which your program is dependent, and which to link with program.
LIBCC		-lstdc++	C++ i/o stream and basic class library.
LOADLIBES		*any*	Additional standard libraries to link with your program (e.g. -lm).
NODEBUG	no	no	Produce source-level debugging information.
		yes	Don't produce debugging information – unless you add **–g** to CFLAGS.
OBJS		*any*	Optional list of object files which make up the program.

Variable Name	Default Value	Permissible Values	Description
PROFILE	no	no	Do not generate or collect profiling code or data.
		yes	Generate code to collect normal *gprof* profiling data (time in each function and call graph).
		lines	Generate code to collect line-by-line *gprof* profiling data.
		feedback-generate	Generate code to collect profiling information which can be fed back to the compiler.
		feedback-use	Optimize the program using data collected by running a program previously built with feedback-generate.
		gcov	Generate code to count branches, and the extra data required by the *gcov* code-coverage program.
PROG		*any*	Name of final executable file, see previous table. If you are now (or may ever be) using Windows, remember to pick file names which fit within the file extension conventions of the Windows filesystem, and ensure your file names are still unique after ignoring differences between upper and lower-case letters.
RDEBUG	no	no	Don't include standalone remote debug stub.
		yes	Include remote debug stub, see Section 14.4.2 "Serial Debugging with SDE Debug Stub".
		immed	Include stub, and cause breakpoint before calling main().
SBD	NOSBD	various	Target board name: see Chapter 8 "Target Specific Libraries".
SDETOP	../..	*any*	The SDE kit and examples base directory, relative to the example directory – but you can also specify an absolute pathname.
SRCS		*any*	List of source files comprising program.
UNCACHED	no	no	Link the program to run cached.
		yes	Link the program to run uncached – for tracing with a logic analyser, for example.

You should rebuild your program from scratch whenever you change any *makefile* parameter. You can delete the old object files easily by running the command "**sde-make clean**".

You can generate a "standalone" Makefile for any example program which is customised to your selected SBD setting, which may help you to generate your own Makefile when you don't need the full multi-target flexibility of the SDE build system. Do this by running "**sde-make SDEmakefile SBD=***xxx*", and then try it out by running "**sde-make -f SDEmakefile**".

Note that .../sde/examples/make.mk also includes the file .../sde/kit/rules.mk, which defines some additional compilation rules, for example to add support for the ".sx" file extension (which identifies assembler files that need to be passed through the C pre-processor[14]).

[14] Equivalent to gcc's handling of the ".S" extension, but compatible with Windows, which can't distinguish upper and lower-case file names.

Porting an ISO / ANSI C Program

This chapter is intended to help you port an existing C application or benchmark program that is compatible with the C library defined by the ISO C90 or ANSI X3J11 standard, as described in [Kern88]. Most simple, self-contained programs will port with no difficulty. The easiest approach to porting is as follows:

1) Create a new sub-directory in the `.../sde/examples` directory (if you're used to an integrated environment, this subdirectory will be your "project") and put your source code there.

2) Copy the *makefile* from the most similar example and edit that. For integer-only programs copy the *dhrystone* makefile; if it uses any floating point arithmetic, then copy the *whetstone* makefile.

3) Edit the new *makefile* and change the definitions of **`PROG`** and **`SRCS`** to represent your final program name, and the list of object files which make it up. Note that object files have the `.o` extension, not `.obj` or anything else.

4) Check the other *makefile* variables, with reference to Section 9.2 "Example Makefiles". In particular check that the **`FLOAT`** variable is set to either `yes` or `ieee` if your program performs any floating point arithmetic, see Section 9.2 "Example Makefiles" and Section 12.5 "Software Floating Point".

5) If you need to measure the execution time of small sections of your code, then use the `clock()` function, or refer to *elapsed time* below.

6) Make and run your program. You could test it first with the GNU MIPS simulator, as described in Chapter 5 "Quick Start". Don't use a high loop count in benchmarks, as the simulator is not fast (hint: use "`#ifdef __SIM`" to select a smaller loop count). To run it on real hardware, follow the instructions in Chapter 8 "Target Specific Libraries" and Chapter 14 "Debugging with GDB".

7) If you want to create a Makefile which is to some extent independent of the flexible, but complex SDE build system, then you can generate a "standalone" Makefile for your own program (or one of the examples), which is customised to your SBD setting. Do this by running "**`sde-make SDEmakefile SBD=`***`xxx`*", and then use it by running "**`sde-make -f SDEmakefile`**". You can then edit `SDEmakefile` to customise it for your own project.

The obvious portability considerations of byte-endianness and word size shouldn't require any explanation these days. But you should be aware of the following special considerations which apply to programs built with SDE's run-time system, as compared to the environment provided on a full-blown UNIX-like system.

- *File i/o* : other than to or from the console terminal is possible when using an MDI-interfaced probe or simulator, or the GNU simulator, or on boards with network hardware and suitably equipped PROM monitors, see Section 19.1.1 "Remote File I/O". In other cases you will have to compile the data into the program.
- *Time and date* : is returned by the ISO / ANSI `time()` function, but can return only the elapsed time on boards without a battery-backed real-time clock chip; on such boards the first call will return zero.
- *Elapsed time* : can be determined on all supported boards with the ISO / ANSI `time()` and `clock()`, or POSIX `gettimeofday()` functions. The `clock()` function is the easiest to use for benchmarking: it returns the elapsed time in units of 1 μs. But note that unlike POSIX it measures elapsed *real* time, not *cpu* time; in other words it **does** include time spent waiting for console input/output. Be careful to put calls to `clock()` around computational code only. See Section 19.1.7 "Elapsed Time Measurement" for details of the other functions.
- *Signal handling* : is primitive. Since the console is polled, the Ctrl-C interrupt (**`SIGINT`**) will only be detected while you are performing i/o.
- *POSIX termios functions* : and `ioctl` interface are supported, see the <*sys/termios.h*> header file. The older *termio* and *sgtty* interfaces are not supported.

Common problems when converting to MIPS® architecture

These remaining points are general warnings about idiosyncrasies of the MIPS architecture and its compilers, which can cause confusion when porting programs.

- *Unaligned addresses*: will cause an "Address Error" exception (a **SIGBUS** signal). This won't affect most programs since the compiler correctly aligns structure fields unless specifically instructed otherwise, see Section 12.4 "Unaligned Data". The `malloc()` family also aligns all requests to an 8-byte boundary (the maximum ever required by the CPU). But beware when type-casting pointers to small types into pointers to larger types (you can try using the compiler's **–Wcast–align** option to catch these).

  SDE includes an exception catcher and emulator for unaligned loads and stores; you just have to call the function

  ```
  _mips_unaligned_init()
  ```

  at the start of your program to install the handler, or simply define "`FEATURES=unaligned`" if you are using the example makefiles. But it's not fast; *don't* use it for benchmarks, and don't use it for a real application unless the unaligned references are very infrequent.

- *Null pointer references*: will cause a "TLB Miss" exception (a **SIGSEGV** signal), unless you set up a dummy TLB mapping for address 0. Memory is normally accessed through the cacheable KSEG0 or uncacheable KSEG1 address spaces, which begin at `0x80000000` and `0xa0000000` respectively.

- *Use of "short" variables*: often prevalent in programs written for 16-bit or x86 processors, generates inefficient code on MIPS architecture processors, particularly if used for for loop counters and array indices. There are no MIPS instructions which operate on sub 32-bit values, and they have to be synthesised from multiple instructions. Although the compiler attempts to avoid excessive conversions, always use "*int*" for such purposes, unless you specifically need the semantics of 16-bit arithmetic.

- *Character signedness*: ISO and ANSI C permits *char* variables to be implemented as either signed or unsigned – it's compiler dependent. MIPS compilers historically made "*char*" variables default to *unsigned* (because it makes faster code); if your program has been developed in a context where those variables were signed, it may not work correctly on MIPS; you may get caught out by mistakes like assigning the integer result of `getc()` to a char variable, and then comparing that with EOF(integer –1).

  You can specify "*signed char*" explicitly for individual variables – which will make your code more portable. But if it is deeply ingrained in your application, then you can use the compiler's **–fsigned–char** option, which changes the default.

- *Bitfield signedness*: Some compilers arbitrarily treat bitfields as implicitly unsigned, but this is not the case for GCC, which uses your type definition as written. But accessing signed bitfields generates slower code, especially when using the MIPS16 ASE. You can either modify your structure definitions to add explict "*unsigned*" type qualifiers, or change GCC's default behaviour using its **–funsigned–bitfields** option.

- *Small variables*: of 8 bytes or less are stored separately from larger variables, to allow them to be accessed more quickly. This can cause strange link-time errors if you have not declared your global variables consistently in all modules ("relocation truncated" is the usual one). See Section 12.3 "GP-relative Addressing" for more information.

Standard Libraries

11.1 ISO / ANSI C Library

SDE's C library (`libc.a` and specified to the linker as **–lc**) follows the ISO C Standard (ISO 9899:1990[1992]), also known as ISO C90, and formerly the ANSI X3J11 committee's standard for the C programming language, It has been validated using the Plum Hall Validation Suite. The full ISO/ANSI specifications are long and careful, so this section lists only differences from the standard as described in Appendix B of *The C Programming Language* by Kernighan and Ritchie [Kern88] – yet another reason to invest in that essential volume.

Note that a number of the functions in the C library assume the existence of a POSIX-like "operating system" interface, which is not included as part of the C library. The notable omissions are listed below, and one possible implementation of them is contained in the embedded system kit, which can be used "as-is", or modified or even completely replaced – to suit your particular requirements.

Input and Output: <stdio.h>

All functions are supplied. However, the *stdio* functions in the library themselves call externally supplied low-level POSIX file i/o primitives. If your program is running on one of the boards supported by SDE's run-time system, then it contains "drivers" which implement the file i/o primitives . If not, or if you don't want to use our kit, then you will have to provide these routines yourself. They must have the standard POSIX semantics:

```
int      open (const char *path, int flags, .../*int mode*/);
int      close (int fd);
ssize_t  read (int fd, void *buf, size_t n);
ssize_t  write (int fd, const void *buf, size_t n);
long     lseek (int fd, long off, int whence);
int      fstat (int fd, struct stat *stb);
int      ioctl (int fd, unsigned long cmd, ...);
```

The *stdio* functions only support the UNIX-style line ending convention, e.g. `'\n'` is always written as a single line-feed character. The ISO / ANSI-specified `"b"` mode can be given to `fopen` etc., and this is passed to `open` as the `O_BINARY` flag bit. It is then up to the `read` and `write` "system calls" to do any translation that might be required.

Character Class Tests: <ctype.h>

All functions are supplied.

String Functions: <string.h>

All functions are supplied.

Mathematical Functions: <math.h>

All ANSI C floating-point functions are supplied (with additions from IEEE–754), in a separate maths library named `libm.a`, and specified to the linker as **–lm**. This library is based on code developed at the University of California, Berkeley. We have assembler-coded some key functions (`drem`, `rint` and `sqrt`). There are two additional, non-standard functions which accept and return single-precision floating point values, namely:

```
/* single-precision square root */
float    sqrtf (float);

/* single-precision absolute */
float    fabsf (float);
```

Utility Functions: <stdlib.h>

All functions are supplied.

The *malloc* family requires an external function with which to obtain sequential, contiguous blocks of memory:

```
void *   sbrk (int nbytes);
```

Note that `nbytes` may be negative if memory is being returned to the "system" from the end of the memory pool (although this is not used by the existing *malloc*). A rudimentary implementation of `sbrk` is supplied in our standard run-time system.

Diagnostics: <assert.h>

Supplied.

Variable Argument Lists: <stdarg.h>

Supplied, together with the old <*varargs.h*> version.

Non-local Jumps: <setjmp.h>

Supplied.

Signals: <signal.h>

These functions are not implemented in the C library itself, as they are operating-system dependent. The header file is present, and a simple implementation of the POSIX *signal* handling functions is provided in our standard run-time system, see Section 19.1.6 "Signal Handling".

Date and Time Functions: <time.h>

All functions are supplied, except for the hardware dependent `clock()` and `time()` functions, which are implemented in our standard run-time system, see Section 19.1.7 "Elapsed Time Measurement".

Implementation-defined Limits: <limits.h> and <float.h>

Supplied.

11.1.1 ISO C99 library support

Support in the SDE C library and associated header files for the new ISO C99 standard is by no means complete, but the C99 <*stdint.h*> and <*inttypes.h*> header files are provided, and the `printf()` and `scanf()` family of functions support the new C99 formatting codes. There are likely to be more C99 features appearing in future releases.

11.1.2 Thread Safety

The SDE C library can be made fully thread-safe and reentrant, using the *SDEthreads* API to protect shared data and manage thread local storage. This API is defined by the header file <*sdethread.h*>. Any RTOS wishing to use the SDE libraries in a thread-safe manner must implement a simple glue or "shim" layer, mapping from the SDEthreads API to its own primitives. A dummy version of the SDEthreads API, suitable for single-threaded code only, is provided in the file `.../sde/kit/share/stubs.c`, and can be used as a model.

MIPS Technologies offer a number of Thread Support Packages (TSPs) which integrate popular RTOSes with SDE – contact us for the current list.

11.1.3 Minimal C library

If program size is critical, and you do not need access to the full-blown library facilities, then you can significantly reduce the amount of the C library that gets linked into your program by avoiding the use of the high-level *Input and Output* functions described above. To output console messages in this case you must call only the functions `_mon_putc()`, `_mon_puts()` and `_mon_printf()` functions, which have identical interfaces to their *stdio* equivalents, except that they talk directly to the PROM monitor or your hardware; also the `_mon_printf()` function does not support floating point. For console input you can use `_mon_getc()` to read a single character at a time.

When your application is known to have limited requirements for its runtime environment, you can reduce the code size further by adding the some of the following definitions to your application *Makefile*.

- If your application does not need de-initialization features like `atexit()`, and a simplified stacktrace when unhandled exceptions occur, then use

  ```
  CRT0FLAGS += -DMINKIT
  ```

- If your application doesn't use exceptions, you can avoid the inclusion of exception handlers with

  ```
  CRT0FLAGS += -DSMALLXCPT
  ```

  Note that this disables handling of all sorts of exceptions, including those caused by hardware faults.

- If your application doesn't use constructors or destructors, you can disable their support with

  ```
  LDFLAGS += -nostartfiles
  CRT0FLAGS += -DNOCTORDTOR
  ```

  Note that some third party libraries may rely on the availability of this feature.

- If you don't need any of the optional kit FEATURES, then you can disable the initialization code via

  ```
  CRT0FLAGS += -DNOFEATUREINIT
  ```

11.2 IEEE-754 Floating Point Emulation Library

SDE's floating point emulation library is named `libe.a`, and specified to the linker as **–le**. It implements single- and double-precision IEEE–754 floating point, but using only integer instructions. It is invoked either directly by subroutine calls from your program (if you specify the **–msoft–float** compiler options), or from a trap-based FPU instruction emulator (to fix up exceptional conditions, or when your code was built for a hardware FPU which is absent).

There is no external documentation for this library, other than the header file <*ieee754.h*>. See Section 12.5 "Software Floating Point" for further usage information.

The library includes two copies of the same code, compiled with different options:

1) A pedantic emulation of the MIPS floating point unit, which is used to implement the trap-based FPU hardware emulation. This uses function names like `ieee754dp_add`.
2) A soft-float version which will be invoked by compiler-generated subroutine calls when compiled with the **-msoft–float** option. This version of the library has been tuned for speed by removing support for floating point exceptions, flag bits, and rounding modes other than "round to nearest". These functions have names like `__adddf3`.

You'll find a primer on floating point and its implementation in the MIPS architecture in [Sweet99].

11.3 Multilibs

SDE can generate code for a large range of MIPS ISAs, and variants such as endianness, register size, soft/hard floating point, an so on. See Chapter 12 "Compiler Options" for a full description of the MIPS-specific compiler options.

In order to support this the standard libraries are supplied in many different flavors, organised into directory hierarchies below `.../sde/lib` and `.../lib/gcc/sde/`*`compiler-version`*. This mechanism is known as *gcc multilibs*, and when you link your program using the *sde−gcc* front-end, it automatically determines the directories which contain the libraries that match the compiler architecture flags that you specified.

As long as you use *sde−gcc* front-end to link your program you don't really need to know how the library directories are organised. But if for some reason you need to use the raw linker (*sde−ld*), or you're just curious, then use this command:

```
$ sde-gcc [your options] --print-multi-directory
```

That will display the directory below `.../sde/lib` which holds the libraries which match your particular group of options. There may be no directory for combinations of options which don't make sense. The set of options which effect the choice of multilib are currently: **−EB/−EL**, **−mips64/−mips32r2/−mips32**, **−mips16**, **−mhard−float/−mfp64/−msoft−float/−mno−float**, and **−mno−data−in−code**.

11.4 Library Source Code

Customers who purchase the MIPS® Software Toolkit receive all of the libraries as source code, as well as in pre-compiled form. Most users will never need to recompile the libraries themselves, but the option is available in case you need to modify a library function, or build debugging or profiling versions of the libraries.

To rebuild the libraries simply change directory to the root of the library source code, and run *sde-make*, like this:

```
$ cd .../sde/libsrc
$ sde-make
```

That will build the C library, maths library, and floating point emulation library in sub-directories `c/OBJ`, `math/OBJ`, and `ieee/OBJ` respectively. All supported *multilib* combinations will be built.

You can also override some of the compiler options like this:

```
$ sde-make DEBUG="-O0 -g" clean all
$ sde-make DEBUG="-pg" clean all
$ sde-make DEBUG="-pg -g" clean all
```

In the first case you'll build a "debuggable" version of the libraries, in the second a profiling version, and in the third case a profiling version with line-number information.

Finally you may want to install all of your newly built libraries, replacing the pre-built libraries that were supplied as part of SDE.

```
$ sde-make DESTROOT=/home/joe/sde-6.06 install
```

But beware: that will overwrite all of the supplied libraries, so you might want to make a copy of the original SDE libraries first, for safe keeping, e.g.:

```
$ cd /home/joe/sde-6.06/sde
$ tar cf - lib | gzip -9 >lib-orig.tgz
```

Compiler Options

The ‘‘MIPS Options’’ section in the GCC manual lists those compiler options which are specific to MIPS-Based processors. This chapter provides some more explanation about these options, and how you might use them.

12.1 Architectural Flags

There are several flags which adjust the class of instructions generated by the compiler or assembler to match your particular CPU type. You can get more information about the architectural features and choices mentioned here in [Sweet99].

12.1.1 Endianness Flags

The most fundamental architectural switch controls whether to generate big-endian or little-endian code. MIPS architecture processors may be configured either way, but the rest of the hardware usually determines which way your system must work. Software has to be compiled to match the way the CPU is configured, or it will fail every time you perform a sub-word load or store.

It is possible to write bi-endian code by very careful assembler coding (e.g. by performing all data accesses as aligned word transfers), but this is likely to be required for only the first few instructions after a hardware reset, until you have configured the CPU and/or device endianness correctly.

–EB Generate code and data for a big-endian CPU.

–EL Generate code and data for a little-endian CPU.

12.1.2 Instruction Set Flags

SDE supports all official and currently implemented 32- and 64-bit MIPS instruction set architectures (ISAs). But the compiler will only generate code compatible with the base MIPS32 ISA unless one of the following switches is used:

–mips1 Issue instructions from the original MIPS I ISA. Compiler/assembler only – no libraries are provided.

–mips2 Issue instructions from the MIPS II ISA (branch likely; square root; 64-bit floating point load/store; faster floating point truncate). Compiler/assembler only – no libraries are provided.

–mips3 Issue instructions from the MIPS III ISA (64-bit instructions; 32 f.p. registers). See Section 12.6 ‘‘64-bit Support’’ for more information. Compiler/assembler only – no libraries are provided.

–mips4 Issue instructions from the MIPS IV ISA (floating point multiply-add/sub, indexed addressing, reciprocal, etc.). Compiler/assembler only – no libraries are provided.

–mips5 The MIPS V ISA introduces a set of ‘‘paired single’’ floating point operations which work in parallel on two single-precision values packed into one register, offering a full range of dual operations. Compiler/assembler only – no libraries are provided.

–mips32 The new, rationalised, 32-bit MIPS32 instruction set defined by MIPS Technologies in 1998/99. It’s not really very different from **–mips2**, but it picks up the conditional move instructions and rationalises the integer multiply/accumulate instructions (which were formerly CPU-specific). The ‘‘branch likely’’ instructions are officially deprecated in MIPS32, but the compiler will still generate them when tuning for CPUs for which it knows they don’t have an adverse performace impact. This is the default ISA if no other architecture flags are used.

–mips64 MIPS Technologies’ rationalised 64-bit MIPS64 instruction set, which is a superset of both **–mips4** (at the user level) and **–mips32**.

–mips32r2

An update of the specifications added some useful new features to the MIPS32 ISAs in September 2002. Many of these features are for the OS only; but there are also a few new user-level instructions:

- *Bit-rotate* : previous MIPS ISAs had only shifts. The compiler will make use of the hardware rotate instruction if your source code is written so as to perform the rotate in a single expression. For example:

```
unsigned int a, b, r;
/* fixed rotate right by 8, or left by 24 */
b = (a >> 8) | (a << 24);
/* variable rotate right */
b = (a >> r) | (a << (32 - r));
/* variable rotate left */
b = (a << r) | (a >> (32 - r));
```

- *Bit-field operations* : single-instruction unsigned bitfield extract and insert instructions make for more efficiency when doing just that... Note that *gcc* treats bitfields as signed if you don't use an explicit `unsigned` type modifier – use the **–funsigned–bitfields** option to change that behaviour. The compiler will sometimes use them when given simple and obvious mask and shift expressions. In cases where it doesn't you can use the explicit insert/extract intrinsics described in Section 18.2 "Intrinsics for MIPS32® Architecture".
- *Byte-swap instructions* : the new instructions **`wsbh`**, **`dsbh`** and **`dshd`** swap bytes within halfwords, or halfwords within doublewords, in a register. So you can do a full 32-bit or 64-bit byte-swap in just two instructions. The compiler will not generate these instructions automatically, but you can access them via intrinsics defined in Section 18.2 "Intrinsics for MIPS32® Architecture".
- *Sign-extend instructions* : bytes and 16-bit values can already be sign-extended automatically when loaded from memory; these new instructions improve code for data which is already in registers.
- *64-bit FPU* : a MIPS32 Release 2 CPU may be paired with a 64-bit FPU, and the extra 16 registers will be used by the compiler if you give it the **–mfp64** option.

–mips64r2

The Release 2 update to MIPS64 adds the same new instructions as MIPS32 Release 2, with additional 64-bit variants.

Once you've defined your base instruction set, there are a collection of "instruction set extensions" which you can enable:

–mips16 Compile using the MIPS16 "ASE". Each MIPS16 instruction is only 16 bits in size, and although a compiler must use more MIPS16 instructions to compile a function than would be required with the MIPS32 ISA, it allows simple integer code to be compiled with a 30-40% saving in space.

Use of this option is a decision with lots of consequences: see longer discussion in Section 12.7 "MIPS16® ASE support" below.

Warning: although the name "MIPS16" seems to fit in with "MIPS32" and "MIPS64", it really is something quite different. In fact, MIPS16 encodings are available for 64-bit instructions too.

The MIPS16 ASE is not available on all CPUs. It also isn't possible to write a complete system using MIPS16 instructions, since some vital instructions (CPU control, floating point, etc) have no MIPS16 encoding.

MIPS16 instructions will probably only ever be generated by compiled code, so you will only ever see assembler code when looking at disassemblies or compiler intermediate files. In assembler source files you'll see that assembler code must request generation of MIPS16 code using an explicit '`.set mips16`' or '`.set mips16e`' directive; the command line option is not passed to the assembler by the `sde-gcc` front end.

–mips16e The MIPS16e ASE is an extension to the MIPS16 encodings, built on the basis of experience with some large programs and achieving a useful improvement in density with a few extra instructions. This

variant is standard on MIPS32 CPUs; in fact, the combination of flags "**–mips32 –mips16**" implies **–mips16e**.

–msmartmips
This option is only valid if you've selected a MIPS32/MIPS64 instruction set, and SmartMIPS cores always implement MIPS16e too. It allows the toolchain to exploit the SmartMIPS extensions to the base MIPS32 ISA: in particular the indexed load (used with grateful thanks by the compiler) and enhanced multiplier instructions – the latter available only through assembler code or special C intrinsics, see Section 18.7 "Intrinsics for SmartMIPS® ASE".

SmartMIPS CPUs also anticipate the bit-rotate instruction from MIPS32 Release 2, as in **–mips32r2** above.

–mpaired–single
For the MIPS32 Release 2 and MIPS64 ISAs only, where 32×64-bit floating point registers are enabled (i.e. **–mfp64**), this flag enables use of the "paired single" SIMD floating point extension which provides instructions to do two single-precision (32-bit) floating point operations at once, keeping the operands in pairs within a 64-bit register. More details on this option can be found in the [Gcc] manual.

–mips3d
Enables the MIPS-3D ASE which includes additional paired-single instructions that are designed to improve the performance of 3D graphics operations. Implies **–mpaired–single**.

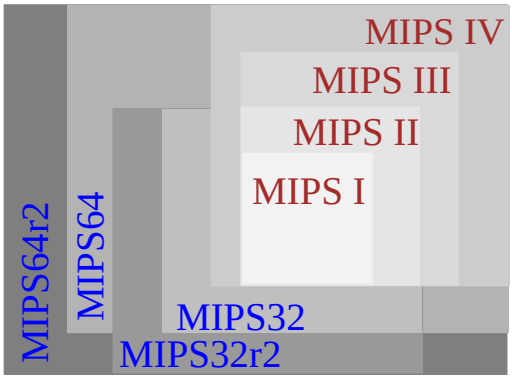
–mdsp This option is only valid if you've also selected the MIPS32 Release 2 instruction set. It tells the compiler to allow the use of the MIPS DSP ASE either automatically where possible, by using vector types, and by use of builtin intrinsics, as described in Section 18.10 "Intrinsics for MIPS® DSP ASE". The ASE is also enabled if the **–march=** option specifies one of these CPUs: `24ke`, `24kec`, `24kef`, `34k`, `34kc`, `34kf`, `74kc`, `74kf`, or `74kx`.

–mno-dsp Prevents the compiler from generating MIPS DSP instructions, even if the selected CPU architecture would support it.

–mmt This option is only valid when you've selected the MIPS32 Release 2 instruction set. It has no direct effect on the compiler, but instructs the assembler to allow the the MIPS MT ASE instructions. These instructions can be generated from C code by using the intrinsics described in Section 18.9 "Intrinsics for MIPS® MT ASE".

A CPU which supports a given ISA will happily run code compiled for the previous variants with which it's backwardly compatible:

Figure 12-1 Relationship of MIPS® ISAs



In practice the first criteria for choosing which level to go for is whether you want to use 64-bit integer data types, which are available only with **–mips64** and **–mips64r2**.

Once you've chosen the integer data width, you'll get small performance increments by choosing the most specialised (usually highest-numbered) instruction set which matches your CPU; you'll make your binary program

more portable by using the lowest number.

The MIPS32 instruction set (or its Release 2 variant) is usually best for applications which don't use 64-bit integer variables, and which don't use floating point heavily – even when you've got a 64-bit processor, since you don't waste data cache space storing unnecessary sign extensions.

12.1.3 CPU Flags

The target CPU type may be specified using the compiler's **–mtune=** option. This allows the compiler to optimize the scheduling of instructions to match your CPU's pipeline. If it is not specified, then the compiler picks the most generic CPU type which matches your requested instruction set (e.g. 4Kc® for **–mips32**), but this may generate sub-optimal code for faster CPUs.

Specifying the CPU type also allows the compiler to make more intelligent choices about CPU-specific features, such as the optional presence of fast or slow multipliers, etc.

In addition to this, the **–march=** option may be used to specify the precise set of instructions and features provided by the target CPU. It also also selects the pipeline scheduling parameters if **–mtune=** is not used explicitly. For compatibility reasons the current SDE makefiles do not make use **–march=**, but use **–mips32**, **–mips32r2**, etc, to select the base ISA – this is likely to change in a future release.

Table 12-1 List of –mtune= names

–mtune=	–mips			Comments
	32	32r2	64	
4km, 4kc	✓			32-bit synthesisable 4Kc and 4Km cores, with fast multiplier
4kp	✓			32-bit synthesisable 4Kp core, with slow multiplier
4kem, 4kec	✓	✓		32-bit synthesisable 4KEc and 4KEm cores, with fast multiplier
4kep, m4k	✓	✓		32-bit synthesisable 4KEp and M4K cores, with slow multiplier
5kc, 5kf	✓		✓	64-bit synthesisable 5K core family; the 5Kf core has an FPU
20kc	✓		✓	64-bit 20Kc hard core
24k, 24kc, 24kf	✓	✓		32-bit synthesisable 24K core family; the 24kf option tunes for a 64-bit FPU running at half the integer pipeline frequency.
24kx	✓	✓		Specifies a 24Kf core, but with the FPU configured to run at the full integer pipeline frequency.
24ke, 24kec, 24kef, 24kex	✓	✓		Enhanced version of the 24Kc and 24Kf cores, with additional features such as the MIPS DSP ASE. The 'x' suffix specifies a full frequency (1:1) FPU.
34k, 34kc, 34kf, 34kx	✓	✓		32-bit synthesisable 34K core family, which supports the MIPS MT and MIPS DSP ASEs.
74kc, 74kf, 74kx	✓	✓		32-bit synthesisable, superscalar 74K core family, which supports revision 2 of the MIPS DSP ASE. The 'x' suffix indicates a full frequency FPU.
25kf	✓		✓	64-bit 25Kf hard core

Other CPU-specific options

You can control some features at a still finer level where necessary:

–mbranch–likely=yes
Enable ‘‘branch likely’’ instructions with **–mips32** and **–mips64**, even though they are officially deprecated.

–mbranch–likely=no
Don’t use ‘‘branch likely’’ instructions.

–mbranch–likely=predict
Only use ‘‘branch likely’’ instructions if the compiler predicts that the branch is ‘‘very likely’’ to be taken.

–mcheck–range–division
Generate code to check for integer divide overflow – range checking is disabled by default.

–mno–check–zero–division
Don’t generate code to check for integer divide by zero – checking is the default, except with **–mips16**.

–mhard–float
Emit hardware floating point instructions – this is the default.

–msoft–float
Emit calls to a software floating point emulation library.

–mno–float
This option is treated by the compiler’s code generator as equivalent to **–msoft–float**, i.e. any use of floating point will generate calls to emulation functions, however it also instructs *sde–gcc* to link your program with libraries which do not include those emulation functions (thus causing a linker warning if they are called) and which also omit all hidden floating point support code, such handling of floating point format codes in `printf`() and `scanf`().

–mfp64
Emit hardware floating point instructions for a 64-bit FPU – this is the default for 64-bit ISAs, but can also be used in conjunction with **–mips32r2**, which allows a 64-bit FPU to be paired with a 32-bit integer ALU.

12.2 Optimization Options

The ‘‘Optimize Options’’ section in the GCC manual lists the various optimization techniques that are available; serious users should read that. But it’s traditional to provide numeric options – the higher the number, the more optimization. You should never compile without at least **-O** (equivalent to **-O1**) unless you’re debugging; GNU C’s un-optimized code is *really* unoptimized. Serious application code will be compiled with at least **-O2**; higher numbers may make your code bigger, and the trade-offs are discussed below.

With GCC each number adds more optimization techniques, while keeping all the options from the lower numbers. For a detailed list of which optimizations are enabled at each optimization level see the [Gcc] manual, but in summary:

–O0 Do not optimize.

–O, –O1 Optimize by trying to reduce code size and execution time, but without performing any optimizations that take a great deal of compilation time.

–O2 Performs nearly all available optimizations that do not significantly increase code size. In particular the compiler does not perform loop unrolling or function inlining when you specify **–O2**. Compared to the optimizer settings documented in the [Gcc] reference manual we also enable **–fweb** at **–O2**.

–O3 As **–O2** but also enables **–finline–functions –frename–registers –funswitch-loops**.

–Os Optimize for size: a lot like **–O2**, but with aditional optimizations to reduce code size, and disabling **–falign–functions –falign–jumps –falign–loops –falign–labels –freorder–blocks.**

12.2.1 Optimizing for Speed

In our experience maximum performance is usually obtained with **–O2** or **–O3**, possibly with the addition of the **–funroll–loops**. It depends on your application, because sometimes the increased code size caused by loop unrolling and function inlining can slow a program down by evicting useful instructions from of the instruction cache. We suggest experimentation, and using profile feedback to tune the loop unroller (see Chapter 15 ‘‘Profiling with GPROF and GCOV’’).

There are many other compiler flags which allow you to control individual optimizations. Not all of them will do anything useful, but here are a few which do have some useful effect:

–funit-at-a-time

Enabled at **–O2**. Instructs the compiler to perform whole module (intra-module) optimization by completely parsing a source file before beginning optimization and code generation. In this way the compiler can use information about all of the functions in the module to make better inlining and optimization decisions.

Furthermore this can perform ‘‘inter-module’’ optimization of your whole program, or a subset of it. This exposes many more optimization opportunities to the compiler, at the cost of greatly increased memory usage in the compiler and compilation time. This feature only works for C, and not yet C++. It requires changing your Makefiles so that instead of using individual commands to compile each module to object code, and then linking the object files together, like this:

```
sde-gcc -O2 -c moda.c -o moda.o
sde-gcc -O2 -c modb.c -o modb.o
sde-gcc -o prog moda.o modb.o ... -lc ...
```

... you now compile a group of modules together with a single invocation of the compiler, like this:

```
sde-gcc -O2 -c moda.c modb.c -o all.o
sde-gcc -o prog all.o ... -lc ...
```

––param inline–unit–growth=

The automatic function inliner (enabled at **-O3**) can be fine-tuned to limit the total growth of a module due to inlining, as a percentage. For example **––param inline–unit–growth=5** limits the total code growth to approximately 5% – the default being 50%, which may be too high for some embedded applications.

–ffast–math

Switches on **–fno–math–errno**, **–funsafe–math–optimizations**, **–fno–trapping–math –ffinite–math–only** and **–fno–signaling–nans**. Allows the compiler to be much more ambitious when optimizing floating point arithmetic, but it can generate incorrect code if a program depends on an exact implementation of IEEE–754 specifications of precision, non-finites and exception handling. Many embedded applications won’t care about this, and can safely enable these extra optimizations.

–fprefetch–loop–arrays

Enables automatic generation of additional instructions to prefetch data accessed sequentially within loops into the cache. On CPUs which implement the `pref` instruction, such as the 24K and 34K, this can increase performance when accessing large arrays. But since this adds extra instructions it may also reduce performance. You can instead use explicit directives where you know it matters, see Section 18.12 ‘‘Intrinsics for Data Prefetch’’.

–funroll–loops

Unroll loops whose number of iterations can be calculated at compile time, or at run time upon entry to the loop. It also turns on complete loop peeling (see below). This option makes code larger, and may or may not make it run faster. It is enabled automatically by **–fprofile–use**, i.e. when you tell the compiler to perform profile directed optimizations. Most of the loop optimizations can be further fine-tuned using **––param**, see the [Gcc] manual for more details.

–fpeel–loops

Peels loops when there is enough information that they do not roll much (e.g. from profile feedback). It also turns on complete loop peeling which completely removes loops which iterate a small constant number of times. This option is enabled automatically by **–fprofile–use**.

–funswitch–loops
Enabled at **–O3**. Moves loop invariant conditional tests out of the loop, and then duplicates the loop inside each branch of the conditional. For example:

```
for (i = 0; i < n; i++) {
  if (a < 0)
    arr[i]--;
  else
    arr[i]++;
}
```

would become:

```
if (a < 0) {
  for (i = 0; i < n; i++)
    arr[i]--;
} else {
  for (i = 0; i < n; i++)
    arr[i]++;
}
```

–fprofile–generate
Adds code to your program so that when run it will collect profile data which can then be used by **–fprofile–use**. This requires that your program has access to a file system where it can store the profile data. See Chapter 15 "Profiling with GPROF and GCOV" for more details. Your program will run slower with this extra profiling code, so don't use this option when generating your final executable.

–fprofile–use
Use the profile data generated by running a program compiled with **–fprofile–generate** to decide when optimizations which increase the size of a program are worthwhile. This also enables **–funroll–loops –fpeel–loops –ftracer –fprofile–values** and **–fvpt**.

When using optimizations which increase code size we strongly recommend that you measure the effect of each option on your performance.

12.2.2 Optimizing for Size

Use the **–Os** flag to tell the compiler that your priority is to reduce code size. This is similar to **–O2**, but subtly alters optimization heuristics in the interests of making your code smaller. (Higher optimization levels can otherwise increase code size in order to achieve better performance).

Further space savings can be made by the addition of some or all of the following flags. But here as elsewhere: if you're not quite sure what they do, don't use them. Most applications will do just fine with **–Os**.

–finline–functions
Inlining of very small functions can actually reduce code size, by removing the function call overhead. This is now enabled by default by **-Os**, with the following additional parameters implied:

––param inline–unit–growth=0
Limits the total growth of a module due to inlining to approximately 0% – the default for speed is 50%.

––param max–inline–insns–auto=5
Sets the maximum size of function (in internal gcc instructions) which will be considered for automatic function inlining to 5 instructions, – the default for speed is 120 instructions.

––param max–inline–insns–single=5
Similar, but applies to functions declared with an explicit `inline` and to C++ class methods – the default for speed is 500 instructions.

–fmerge–all–constants
Used in addition to the default **–fmerge–constants** this enables merging of *const* variables, as well as constant strings and literals. Languages like C and C++ require that each non-automatic variable has a unique address, so using this will result in non-conformant behavior – you will need to check that your program can survive

this.

–mno–check–zero–division
Prevents the normal insertion of inline code which checks for integer divide-by-zero etc.; this won't affect performance, but it is not recommended when debugging your program. This is the default when compiling for MIPS16.

–fno–rtti
For C++ programs which do not use *dynamic cast* and *typeid*, use this option to disable generation of C++ runtime type identification information for every class with virtual functions. This can reduce the size of the code and data.

–fno–exceptions
For C++ programs which do not use exceptions, use this option to disable the generation of the frame unwind information – which will signficantly reduce the read-only data size.

–ffunction–sections
Causes each function to be emitted into its own unique object code section. See below how this can be used to reduce code size.

–fdata–sections
Like **–ffunction–sections**, but for variables.

Finally, if you are using the standard SDE run-time board support "kit" code, then you can in some cases use a stripped-down version of this library, see Section 11.1.3 "Minimal C library".

12.2.2.1 Code and data garbage collection

You can use **–ffunction–sections** and **–fdata–sections** to reduce the size of some applications, to allow automatic removal of unused functions and variables. But note that if your application does not contain much unused code or data, then these flags might slightly *increase* the total size, due to extra padding between functions and variables.

The trick is achieved by compiling your source files with one or both of these options, which causes each function and variable to be placed into a unique object code section, and then instructing the linker to "*garbage collect*" unused sections, as identified by peforming a tree-walk of all code and data cross-references, starting from the program's entrypoint. The linker will do this when given the **–gc–sections** option.

Here's an example showing just two files being compiled and linked:

```
$ sde-gcc -Os -ffunction-sections -fdata-sections -c a.c -o a.o
$ sde-gcc -Os -ffunction-sections -fdata-sections -c b.c -o b.o
$ sde-gcc -Wl,-gc-sections -o prog a.o b.o
```

If you are using the SDE example makefiles you can do this by setting the CFLAGS and LDFLAGS variables, e.g.

```
$ sde-make SBD=MSIM32 CFLAGS="-Os -ffunction-sections" \
     LDFLAGS="-Wl,-gc-sections"
```

Note that these options shouldn't be used when debugging your code – the multiple sections will confuse the debugger – only do this for production builds. Also see Section 16.4.4 "Linker Garbage Collection" for more details about controlling the linker's behaviour.

Tip: It may be counter-productive to use **–fdata–sections** when compiling MIPS16 code, since it disables the MIPS16 "section-relative addressing" optimization.

12.3 GP-relative Addressing

The GCC manual describes the **–G***num* option, which controls the maximum size of global and static data items that can be addressed in one instruction instead of two. The default value is 8 bytes, which is large enough to hold all simple scalar variables.

This optimization technique is known in MIPS toolchains as *gp-relative* addressing, and relies on the compiler, assembler, linker and run-time initialization code cooperating to pool all of the ‘‘small’’ data items together into a single region, and then setting the *gp* register to point to the middle of this region. These items can then be referenced with a single instruction, using a signed 16-bit offset (i.e. –32768 to 32767) from the *gp* register (*$28*), instead of the usual two instruction sequence. However there are some potential pitfalls with this technique:

- You must take special care when writing assembler code to declare global (i.e. public or external) data items correctly:

  a) Writable, initialised data of *gnum* bytes or less must be put explicitly into the `.sdata` section, e.g.:

  ```
          .sdata
  small:  .word   0x12345678
  ```

  b) Global *common* data must be declared with the correct size, e.g:

  ```
  .comm   small, 4
  .comm   big, 100
  ```

  c) Small external variables must also be declared correctly, e.g:

  ```
  .extern smallext, 4
  ```

- In C you must declare global variables consistently in all modules which define or reference them. For external arrays either omit the size (e.g. `extern int extarray[]`), or give the correct size (e.g. `int cmnarray[NARRAY]`). Don’t just give a dummy size of 1. Watch out particularly for use of the magic compiler/linker variables like `_end`, `_edata`, etc.: they should be declared as character arrays of unknown size, e.g.

  ```
  extern char _end[];
  ```

- If your program has a very large number of small data items or constants, the **–G 8** option may still try to push more than 64KB of data into the ‘‘small’’ region; the symptom will be obscure relocation errors (‘‘relocation truncated’’) when linking. Fix it by disabling gp-relative addressing with the **–G 0** option; most of the time you won’t lose too much.
- Some real-time operating systems and PROM monitors can be entered by direct subroutine calls, rather than via a ‘‘system call’’ trap. The use of simple subroutine calls between sections of the program which were not linked together means that it is not possible for the application and the monitor to share a *gp* area. In this case either the application or the monitor/RTOS (but not necessarily both) must be built with **–G 0**.

When a particular **–G** option has been used for compilation of any set of modules, then it is usually necessary that all other modules and libraries should be compiled with the same value, to avoid linker relocation errors (e.g. one module references a variable which it thinks is in a ‘‘small data’’ section, while the other defines it in a non-small section). To avoid relocation overflow errors when linking, the safest solution is to compile all modules within a mixed 32-bit and MIPS16 system using the same value of **–G**.

Of course larger values of **–G** *num* can be used to increase the scope of this optimization. However, at the moment the only way to find the limit is an iterative process of recompiling with increasing values, until you overflow the 64K limit. One day it may be possible to determine an optimal value automatically.

12.4 Unaligned Data

The standard MIPS load and store instructions require that all data is aligned on its ‘‘natural’’ boundary, i.e *shorts* on a multiple of 2 bytes, *ints* on a multiple of 4, and *doubles* on 8. If the alignment is not correct, then the CPU will generate an address exception.

Because of this restriction, *gcc* will normally align all data structures and their fields on their natural boundaries. However some software ported from 8 or 16-bit CPUs may rely on data structures whose fields align to a smaller boundary (e.g. for network protocol headers, or printer font cartridges, etc.). There are two ways to convince *gcc* to change its default alignment rules:

1) Using the `attribute((align(x))` or `attribute((packed))` mechanisms on individual structure fields; see the *Extensions* section in the GCC manual for details.

2) Precede the definitions of the critical structures with the single line `#pragma pack(x)`, where `x` is the alignment boundary, in bytes. Follow the declaration with the line `#pragma pack()`, which restores the normal alignment rules – don’t forget this, your code will probably continue to work, just get much bigger and slower!

3) In desperation you can compile your program with the **–fpack–struct** option, which removes padding from all structures. But that will make everything slower, and may well cause other incompatibilities.

The compiler will make use of the MIPS unaligned load/store instructions to access unaligned structure fields, but it will result in slower code. So use the `#pragma pack()` control only around critical data structures, and not as a global switch for the whole program.

None of the above solve the problem of unaligned pointers to fundamental types (e.g. `int *`). Currently these can only be handled by installing an exception handler which fixes up instructions that get an *Address Error* (**`XCPTADES`** or **`XCPTADEL`**) exception. So long as you use SDE’s standard exception handlers then you can do this (carefully and slowly) by putting a call to `_mips_unaligned_init()` at the beginning of your code, or simply by defining ‘‘`FEATURES=unaligned`’’ if you are using the example makefile system. But don’t do this if performance is important to you – just use it to get you going when first porting an application.

12.5 Software Floating Point

When an application performs floating point computations and the target CPU is not equipped with a floating point unit (called coprocessor 1, or ‘‘CP1’’ in MIPS-speak), then the floating point operations must be performed by software subroutines. SDE includes an IEEE–754 compliant software floating point library (in library `libe.a`, or **–le** to the linker) which performs floating point arithmetic using only integer operations. There are different ways in which this library gets used:

1) When you use the compiler’s **–msoft–float** option it will keep all floating point values in integer registers (a pair of them for double-precision when using 32-bit registers), and will generate direct calls to the software floating point library to perform all floating point arithmetic. This is the best option if you know that you will never have a hardware floating point unit in your target system.

2) If **–msoft–float** is not used (or **–mhard–float** is) then the compiler will emit code which uses hardware floating point registers and instructions. You then have to include a ‘‘CP1 emulator’’ in your program which catches ‘‘Coprocessor Unusable’’ traps, interprets the instructions, and invokes the software library to emulate them. This results in even slower code than when using **–msoft–float**, but may be the option to use when creating a single program binary which must be capable of working either with or without a hardware floating point unit, detected at run-time. See Section 20.8.1 ‘‘Coprocessor 1 Emulation’’.

Remember that in all cases emulated floating point is *much* slower than hardware – up to 100 times slower for the trap-based emulation.

The example *makefiles* determine which options to use based on the value of the **`FLOAT`** parameter defined for that program, and the **`FPU`** parameter defined for the selected target board. See Section 9.2 ‘‘Example Makefiles’’ and Chapter 22 ‘‘Retargetting the Toolkit’’ for more details.

12.6 64-bit Support

SDE supports the MIPS64 instruction set, which extends the MIPS32 architecture to support 64-bit integer registers. All MIPS64 CPUs, when equipped with an FPU, also support the full set of thirty two 64-bit floating point registers.

There's no "mode switch" for 64-bit operation in MIPS architecture processors. 64-bit CPUs execute all the 32-bit instructions, always producing 64-bit results, and where possible doing the same job – so the **`or`** instruction on 64-bit CPUs is always a 64-bit "inclusive or", and so long as you only give it valid 32-bit operands, you'll always get a valid 32-bit result. Separate 64-bit versions of instructions are required only if they generate results which might overflow from 32 bits (i.e. the 64-bit result might not be equal to the result of sign-extending the 32-bit result to 64-bits); so as well as the 32-bit add **`addu`** there is now a 64-bit **`daddu`**, and similarly for **`dsubu`**, **`dsll`**, etc. See [Sweet99] for an account of how this all works.

When you use the **–mips64** or **–mips64r2** option, you allow the compiler to generate 64-bit instructions, and implicitly select a different *calling convention* – also called an ABI (Application Binary Interface).

12.6.1 64-bit Calling Conventions

Once you select a 64-bit ISA or ABI then your computations will use 64-bit registers, and computations on *long long* variables will use 64-bit machine instructions. To support the wider registers a new and more efficient calling convention is used: by default this is the "N32" ABI.

N32 ABI (default)

Although N32 sounds like a 32-bit calling convention, it requires a 64-bit CPU and is – except for the most trivial cases – wholly incompatible with the old 32-bit MIPS calling convention commonly known as "O32" (which is what Silicon Graphics called it). See [Sweet99] for a discussion of the calling conventions and why they're like they are. You can also find a detailed description of the N32 ABI, and a discussion of 32- to 64-bit porting issues at SGI's MIPSpro™ N32 ABI Handbook.

It is important to note that while the N32 ABI does support 64-bit registers and uses 64-bit instructions, it does not use 64-bit pointers. N32 implements the "ILP32" model, where *int*, *long* and *pointer* types are all 32-bits – only *long long* and *double* are 64-bits. In an embedded system this seems to be the best compromise. It is not clear that 64-bit addressing is useful or sensible in most embedded environments it certainly increases the memory footprint, and it introduces significant portability problems (e.g. `sizeof(void*) != sizeof(int)`) which many embedded applications have not yet had to deal with. If you need to access physical addresses above 512MB then you could use a TLB entry to map the physical address into the 32-bit virtual address space (i.e. KSEG2 or KUSEG); or you could store 64-bit addresses in *long long* variables, and then use assembler subroutines or C `asm`'s to perform the loads and stores.

Unlike some 64-bit versions of GCC, the SDE compiler does not currently provide a 128-bit extended *long double* type when using the N32 ABI: instead a *long double* is treated as identical to a 64-bit *double*. The cost of implementing such an extended precision type would be significantly increased code size, due to the 128-bit software floating point emulation library which would then be required.

N64 ABI (partially supported)

As of SDE v6 the compiler and other tools can generate code which uses the "N64" ABI, which uses 64-bit *long* and *pointer* types, with a 32-bit *int* – known as the "LP64" model. Select this by using the **–mabi=64** option, but note that the N64 ABI is not compatible with N32, and is not currently supported by SDE header files, libraries and run-time system. If you absolutely have to use 64-bit pointers and N64, then you will have to bring your own 64-bit-safe header files and libraries.

O64 ABI (deprecated)

A different 64-bit calling convention was defined and used by Cygnus/RedHat and some of their customers. You could describe it, approximately, as what you get by taking the 32-bit "O32" standard and replacing all the 32-bit fields by 64-bits, and this is called the "O64" ABI (specify **–mabi=o64**). It's incompatible with the SDE libraries, and we have not tested it in any way – we don't recommend its use for new development projects.

ABI-specific Code

If you need to write assembler routines which stand some chance of working in either call-convention universe, use compiler predefined macros as follows:

```
#if _MIPS_SIM == _ABIO32
/* 32-bit O32 calling convention */
#endif
#if _MIPS_SIM == _ABIN32
/* 64-bit N32 calling convention */
#endif
```

For backward compatibility, SDE implicitly sets a pre-processor assertion, which you can test as follows:

```
#if #abi(32)
/* 32-bit O32 calling convention: -mabi=32 */
#endif
#if #abi(n32)
/* 64-bit N32 calling convention: -mabi=n32 */
#endif
```

Also available for backward compatibility are the SGI-inspired `_MIPS_SIM` definitions, as follows:

```
#include <sgidefs.h>
#if _MIPS_SIM == _MIPS_SIM_ABI32
/* 32-bit O32 calling convention */
#endif
#if _MIPS_SIM == _MIPS_SIM_NABI32
/* 64-bit N32 calling convention */
#endif
```

12.6.2 64-bit Optimization

Unfortunately GCC is not always as successful at optimizing 64-bit code as it is with ‘‘normal’’ 32-bit code. In particular it sometimes fails to spot when it can avoid conversions between 64- and 32-bit values in registers, and can fail to optimise certain sub-expressions involving 64-bit constants. We suggest that you use *long long* only where the extra bandwidth or precision is important, and don’t try to use it as a global replacement for *int* or *long*.

12.6.3 64-bit Assembler Changes

Like the compiler, the assembler recognises the directives which identify a 64-bit CPU. See [Sweet99] for a complete description of 64-bit features, or [Kane92] for a reference-manual approach to the machine instructions.

64-bit assembler constants

To prevent uncertainty regarding their size, and whether or not they are sign-extended, immediate operands are truncated to 32-bits. You can specify full 64-bit immediates only for the `dli` instruction and `.dword` pseudo-op.

12.7 MIPS16® ASE support

The ‘‘MIPS16’’ instruction set is an extension to the MIPS architecture (an ‘‘ASE’’) that allows you to build much smaller binaries. It requires that the CPU implement a set of operations encoded with fixed-length 16-bit instructions; this new instruction set is selected with a ‘‘mode switch’’ controlled by a ‘‘least significant bit’’ included in the instruction address. You can successfully build and run a program with a mix of functions built both with MIPS16 and conventional instructions, but you can’t mix the two instruction sets inside one C function.

The MIPS16 ASE is most useful to the smallest and most deeply embedded systems, and is often not implemented on higher-end CPUs.

‘‘MIPS16e’’ is the name of an enhanced version of the MIPS16 instruction set; the enhancements were worked out from experience and help the SDE compiler generate even smaller code. Note that all those MIPS32-compliant CPUs which support the MIPS16 ASE implement the MIPS16e extensions.

Most often a MIPS16 operation corresponds to a single conventional MIPS instruction, but the small size imposes restrictions on choice of registers and the size of ‘‘immediate’’ fields.

For straightforward integer code **–mips16** can cut code size by around one third, but it certainly won’t do this if:

1) you use floating point: the MIPS16 ASE doesn’t encode f.p instructions or registers, which have to be replaced by calls to 32-bit code – even if the CPU has an FPU, or

2) you use unaligned data structures heavily: there are no `lwl/lwr` MIPS16 instructions, so these have to be synthesised as a sequence of byte loads, shifts, ors, etc.

Most users will never, and should never, write MIPS16 assembler code. You’ll find no assembler language documentation here. MIPS16 instructions are meant to be an intermediate code generated by the compiler to save space – possibly at the cost of some speed. MIPS16 CPUs always run the normal 32-bit MIPS instruction set as well, which is usually a better choice for assembler modules.

MIPS16 functions can safely call functions consisting of ordinary 32-bit MIPS instructions, and vice versa. The hardware keeps track of MIPS16 mode by adding a bit zero to the instruction address pointer; so a jump-register instruction to an odd address implicitly switches into MIPS16 mode. Because normal absolute `jal` instructions don’t contain the bottom address bits (since regular MIPS instructions are 4 byte aligned), a new instruction `jalx` is added which calls MIPS16 code from regular 32-bit code, or *vice versa*. The linker automatically converts a `jal` to a `jalx` when it sees a call across the MIPS16/regular-MIPS divide.

MIPS16 functions using floating point must be declared carefully. The compiler automatically generates small ‘‘trampoline’’ stubs to copy floating point arguments and results back-and-forth between ‘‘hard’’ f.p. registers and the MIPS16 integer registers used for f.p. arguments. It’s essential to provide full prototypes for such functions.

Global Variables and MIPS16® code

The global-pointer (GP) optimization used in 32-bit MIPS code to speed up access to small global variables is not usually appropriate to MIPS16[15] code, with its restricted load offsets (all GP-relative addresses would require an extended instruction). A mechanism has been developed for MIPS16 code which accesses variables defined within the same compile unit as the code using short ‘‘section relative’’ offsets. This optimization is of no benefit to ‘‘extern’’ or ‘‘common’’ variables, but is a big win when accessing locally defined variables.

In order for this optimization to be more effective, code compiled using **–mips16** or **–mips16e** will by default also imply the specification of ‘‘**–G0 –fno–common**’’. This has the following implications:

- If you are compiling any modules using a 32-bit ISA, but you expect that they may be linked with MIPS16 code, then you must specify **–G0** explicitly for the 32-bit modules. You can still link with existing, pre-compiled, 32-bit libraries that were compiled gp-relative addressing enabled, so long as the precompiled code does not try to reference global symbols defined in the **–G0** compiled code. The safest solution is to compile all modules in a mixed 32-bit and MIPS16 system using **–G0**.
- The ‘‘traditional’’ (but not ISO / ANSI compatible) C ‘‘common variable’’ behaviour – named after the Fortran construct, which allows several modules to declare the same global variable, as long as no more than one of the declarations actually initialises the variable – will no longer work. If possible you should avoid relying on this feature in portable code, but if it cannot easily be changed in your code, then you will have to specify **–fcommon** on the command line, and you will lose the section-relative addressing optimization on uninitialised global variables (uninitialised *static* variables will be optimized). Existing, pre-compiled libraries which use common variables will continue to work correctly when linked with code compiled with **–fno–common**, as long as they don’t initialise the same variables.
- You can flag individual variables where ‘‘common’’ behaviour is absolutely required, by using *gcc*’s `__attribute__` mechanism. For example:

```
int errno __attribute__((common));
```

[15] Some MIPS32+MIPS16e CPUs have separate instruction and data memories, so can’t embed data in the instruction stream, even for loading constants. The *gp* optimization *is* useful in this case.

Global Register Variables

In MIPS16 code only 8 registers are directly usable for arithmetic and pointers, but the remaining 24 registers are accessible indirectly. The compiler allows MIPS16 code to use gcc's global register variable extension to access these extra registers, which can provide a performance boost for global variables which are very frequently accessed in many separate, small functions. It is recommended that callee-saved registers *$s3–$s7* only are used for this purpose (*$s0* and *$s1* are used by normal MIPS16 code, *$s2* is used by MIPS16 code if there is a hardware FPU, and *$s8* is sometimes used as a stack frame pointer in 32-bit code).

Global register variables must be declared in a header file which is common to all modules, so that the register does not get reused for normal variables or temporaries by 32-bit code. Here is an example of how to declare and use a global register variable:

```
register struct insn *curinsn __asm__("$s3");

unsigned int getinsn_opcode (void)
{
    return curinsn->opcode;
}
```

Divide by Zero Checks (–mcheck–zero–division)

When generating MIPS16 code the compiler will not generate the extra code to check for division by zero, so divide by zero will generate an undefined result. If for debugging purposes you wish division by zero to generate a trap, then use the **–mcheck–zero–division** compiler option.

Execute-only Code / Split I-D RAM

In MIPS16 code the compiler normally places implicit constants inline within the executable code section, interleaved with or following the function which uses them. This allows the constants to be accessed efficiently using the MIPS16 PC-relative load and addiu instructions.

However some MIPS Technologies cores support independent, Harvard-style on-chip instruction and data memories known as SRAM or SPRAM. In such a configuration a program cannot read constant data from the I-side memory without special hardware support[16], which causes the CPU to treat the MIPS16 PC-relative load instructions like an instruction fetch, and ''redirect'' the load from the D-side memory port to the I-side.

Use the **–mno–data–in–code** flag when compiling MIPS16 code to run in ISRAM on a system without the hardware redirect. It will generate larger and slower code (5% larger on average) – so don't use it unless you have to. Also make sure that you use the **–mno–data–in–code** flag when linking your program, to select a compatible *multilib* variant, see Section 11.3 ''Multilibs''.

When the **–mno–data–in–code** flag is used, it also switches off the **–G 0** option – otherwise the default for MIPS16 code – so that it can place the constants into the small data section, and access them via the *$gp* register. You can use the **–G 0** option explicitly to prevent this, but it may increase code size significantly.

The **–mcode–xonly** flag is a weaker alternative for MIPS16 code running in on-chip ISRAM where the system does implement the hardware redirect (e.g. the M4K). The hardware redirect operates only for PC-relative loads, but MIPS16 code can still create pointers to the implicit constants – most obviously to literal character strings – to be used later by conventional load instructions, which would then read the data from the wrong memory. The **–mcode–xonly** option instructs the compiler not to place constant strings and computed jump tables into the code segment, while keeping simple integer and floating point constants inline with the code. This will usually result in only slightly larger code than a standard MIPS16 compilation. All of the MIPS16 libraries are now built with this option.

The **–mcode–xonly** flag may also be useful for cores which implement the SmartMIPS ASE, which provides an extended virtual memory protection model that can mark pages as ''execute-only''. Similar to the I/D redirect above, the MIPS16 PC-relative load represents itself to the TLB as an instruction-fetch so, for MIPS16 code running in

[16] The M4K core implements the ''redirect'' internally; other cores require external logic to do this – ask your SoC designers whether they have implemented this redirect.

mapped space, use **–mcode–xonly** to prevent strings and jump tables from being placed in the executable code section.

Generating MIPS16® code

Add the compiler flag **–mips16** or **–mips16e**, and the module will be compiled using MIPS16 or MIPS16e instructions to generate compact code. The flags are (mostly) orthogonal in effect to other flags which set code generation options.

It goes further than that: the **–mips16** flag used on the *sde–gcc* command line when linking your files will select MIPS16 or MIPS16e libraries.

Back to compilations: sometimes a module might contain functions you want to compress, and some you would rather compile to regular 32-bit instructions – perhaps because the 32-bit instructions will give you better performance[17], or because you need to use instructions that are not available in MIPS16.

The compiler uses the GCC `__attribute__` extension to permit the instruction set to be selected on a per-function basis. For example:

```
__attribute__((mips16)) void smallfunc ()
{ /* generates MIPS16 code */ }

void __attribute__((nomips16)) bigfunc ()
{ /* generates 32-bit MIPS code */ }

void normalfunc ()
{ /* compiled as per command-line flags */ }
```

It is likely that the attribute construct will be hidden by a macro, which can be controlled by an *ifdef*, e.g.

```
#if __mips
#define large           __attribute__((nomips16))
#define compact         __attribute__((mips16))
#else
#define large
#define compact
#endif

compact void smallfunc ()
{ }
```

If the command-line selects **–mips32**, then `__attribute__((mips16))` will generate extended MIPS16e instructions, otherwise it will generate only ‘‘standard’’ MIPS16 instructions. Similarly, if the command-line selects **–mips16e**, then `__attribute__((nomips16))` will generate MIPS32 code.

If you have used the ‘`mips16`’ attribute, but wish to prevent it from taking effect, then compile with **–mno–mips16**.

Sibling call optimization

If you are mixing MIPS16 functions and 32-bit functions in your program, then it is not safe to allow the compiler to perform its ‘‘sibling call’’ optimization, which can replace a call at the end of a function by a jump to the other function. Since there is no **`jx`** instruction to switch from 32-bit to MIPS16 mode, only **`jalx`**, this optimization must be prevented when a 32-bit function calls a MIPS16 function. If it were to occur, then it would result in a error message from the linker when it tried to relocate the jump instruction. You can prevent this optimization from taking place in two ways:

1) Most easily by using the compiler’s **–fno–optimize–sibling–calls** option.

[17] MIPS16 code always takes longer to execute within the CPU, but if instruction fetch bandwidth is the critical determinant of the performance of some piece of code, then the smaller size of MIPS16 code can make it faster overall.

2) At a more fine-grain level, by ensuring that all global MIPS16 functions are correctly declared with function prototypes which include `__attribute__((mips16))` in the function type. It is not necessary to do this for 32-bit functions, since the compiler will never generate sibling calls from MIPS16 functions.

Main differences between MIPS16® and MIPS16e™ code

The new MIPS16e instructions clean up a few wrinkles where the original MIPS16 definition caused the compiler to generate wasteful code. These are:

- An instruction to save registers and do other function entry housekeeping, with a matching instruction to restore registers on function exit. (They only support a 32-bit register model.)
- Instructions which sign- or zero-extend partial-word values in registers.
- Variants of the indirect jump and jal instructions which don't have a visible branch delay slot.

You'll be surprised how much they help.

12.8 Predefined Preprocessor Macros

Your program can detect what sort of CPU and instruction set it is being compiled for by testing a number of predefined C preprocessor macros. For example:

```
#if __mips == 32
#if __MIPSEB
/* big-endian MIPS32 code */
#endif
#if __MIPSEL
/* little-endian MIPS32 code */
#endif
#enfig
```

The full table of predefined macros defined by GCC for MIPS is as follows:

Table 12-2 Predefined macros for MIPS

Macro	*Purpose*
`__mips`	Defined whenever compiling code for a MIPS ISA. Has as its value the selected ISA level, e.g. 1 for **–mips1**, 32 for **–mips32**, and 64 fot **–mips64**.
`__mips_isa_rev`	The ISA revision level – only relevant for MIPS32 and MIPS64 – has the value 1 for the original revision, or 2 for the second revision of the ISA (i.e. **–mips32r2**).
`__mips64`	Defined when compiling for an ISA which supports 64-bit general purpose registers. Not the same as "`__mips == 64`", since it will also be defined for the 64-bit MIPS III and MIPS IV ISAs.
`__mips_fpr`	Specifies the size in bits (64 or 32) of each floating point register, as selected by the base ISA and ABI, or by the **–mfp64** compiler flag.
`__mips16`	Defined when **–mips16** is used to select generation of compact MIPS16 code.
`__mips16e`	Defined in addition to `__mips16` when generating code for the enhanced MIPS16e ASE available with MIPS32 and MIPS64.
`__mips_hard_float`	Defined when generating hardware floating point instructions.
`__mips_soft_float`	Defined when **–msoft–float** is used, and the compiler will generate calls to a software floating point emulation library.
`__mips_no_float`	Defined when the **–mno–float** flag is used, to request libraries without floating point support, to reduce program size. Otherwise equivalent to `__mips_soft_float`.

Macro	*Purpose*
__mips_dsp	Defined when the DSP ASE is enabled, using either thr **–mdsp** compiler flag is specified, or when **–march=** is set to CPU type 24ke, 24kec, 24kef, 34k, 34kc, 34kf, 74kc, 74kf, or 74kx.
__mips_paired_single_float	Defined when **–mpaired–single** is used to enable the ‘‘paired single’’ SIMD floating point extension.
__mips3d	Defined when the **–mips3d** flag is used to enable the the MIPS-3D ASE.
__mips_smartmips	Defined when the **–msmartmips** flag is used to enable the SmartMIPS ASE, or when enabled implicitly because **–march=** is set to 4ksc or 4ksd.
__MIPSEB	Defined when compiling code for a big-endian CPU, i.e. when the **–EB** flag is used.
__MIPSEL	Defined when compiling code for a little-endian CPU, i.e. when the **–EL** flag is used.
__MIPS_ARCH_*CPU*	Where *CPU* is the name specified with the compiler’s **–march=** option, converted to upper case.
__MIPS_TUNE_*CPU*	Where *CPU* is the name type specified with the compiler’s **–mtune=** option, converted to upper case.
__SOFT_FLOAT	Same as __mips_soft_float, for compatibility with previous versions of SDE.
__NO_FLOAT	Same as __mips_no_float, for compatibility with previous versions of SDE.
__pic__	Defined when generating MIPS/abi position-independent code, as selected by the **–fpic** or **–mabicalls** compiler flags.
__PIC__	Equivalent to __pic__.
__SDE_MIPS__	Defined to indicate that the code is being compiled by an SDE configuration of GCC, and the SDE headers and libraries will be available.
_MIPS_SIM	Indicates the ABI or calling convention in use – takes one of the values _ABIO32 (1), _ABIN32 (2), _ABI64 (3), _ABIO64 (4), or _ABIEABI (5).
_MIPS_FPSET	Indicates the number of 64-bit floating point registers available: 16 or 32. This encodes the same information as __mips_fpr above, but in a different way, and is included for compatibility with Irix.
_MIPS_SZINT	Indicates the size in bits of the int type: 32 or 64.
_MIPS_SZLONG	Indicates the size in bits of the long type: 32 or 64.
_MIPS_SZPTR	Indicates the size in bits of pointer types: 32 or 64.

The compiler also makes a number of predefined ‘‘assertions’’ which can be tested at compile-time, however these are deprecated in favor of the more widely supported conventional pre-processor macros and constants.

Insight Graphical Debugger

SDE includes the ‘‘Insight’’ graphical user interface for *sde−gdb*.

Details of how to connect *sde−gdb* to your remote board can be found in Chapter 14 ‘‘Debugging with GDB’’. Please refer to the comprehensive printed or online GDB manual for more information about the GDB command line interface, which you will still need to use for more complex debugging jobs.

The GUI starts by default when you run **sde−insight** – if you want just a command line, use **sde−gdb**. UNIX users must be running a window system – that is, be sitting at an X-server or X-terminal, with the `DISPLAY` environment variable set correctly. Windows user just have to be sitting in front of their PC.

Insight provides a set of debugger windows, including:

- Console Window: for old-fashioned command-line interaction.
- Source Window: to see the source of the program under test, or disassembled instructions, or interleaved source and instructions.
- Locals Window: to watch the value of local variables.
- Stack Window: to show where you’ve come from.
- Register window: to watch or edit register contents.
- Memory Window: to watch or edit memory contents.

And so on. Play with it and you will find more.

There really ought to be more guidance here as to how to use the system. But time caught up with us. A future version of this manual (and online help) will be more useful; meanwhile you can get some guidance from `http://www.redhat.com/docs/manuals/edk/EDK-1.0-Manual/getting-started-guide/gsdebug.html`.

Debugging with GDB

Source-level debugging of an embedded application requires two components. The host debugger *sde−gdb* has access to your source and object files, and understands the structure of your program and data. But to interact with the running software *gdb* needs to be able to read/write memory and registers, and access on-CPU debug functions on your target system.

The connection between GDB and the target will be one of:

a) A connection which exploits an on-CPU debug connection such as MIPS Technologies' EJTAG. This will need a special piece of hardware (a *probe*) connected to the CPU on the board under test, some physical connection to the probe (typically Ethernet, USB or parallel port), and some host software to connect GDB to the probe.

   MIPS Technologies promotes a software interface called ''MDI''; it's a standard interface for the on-host software which connects to an EJTAG probe. The version of *sde−gdb* included in SDE v5 and higher can talk to any MDI-compatible probe software.

   Some EJTAG probe manufacturers don't provide an MDI interface, but are compatible with *gdb*'s standard remote debug protocol (Abatron, for example). Some others have totally proprietary interfaces, in which case they may come with their own proprietary debugger, which may be compatible with the SDE compiler – check with your probe supplier.

b) An ethernet or serial port connection to the target, together with a ''target monitor'' program running on your target CPU. The target monitor is a little ''server'', attached to the host via serial port or network link, which can be requested to inspect or patch memory, to catch exceptions (particularly breakpoint exceptions) and report the application's CPU state.

   MIPS Technologies' YAMON monitor includes a built-in target monitor, which can communicate directly with *sde−gdb* over a serial port. But if your target doesn't have the YAMON monitor (or if your application takes over exception handling from the ROM monitor, or if you need multi-thread support) then you can instead rely on linking your application with the ''remote debug stub'' code provided with SDE run-time software.

c) Your target may not be a real piece of hardware, but a software simulator. The basic GNU MIPS simulator included with SDE is built-in to *sde−gdb*; while MIPS Technologies' MIPSsim simulator (much more grown-up and accurate) is supplied as a separate DLL which connects to *gdb* via the MDI interface.

Usually you will use *gdb*'s `load` command to download your application to the target – but that can be very slow and tedious over a serial port. If you don't have a dedicated debug probe, then a ROM monitor which supports Ethernet downloading (such as the YAMON monitor) can be very helpful – see Chapter 17 ''Manual Downloading''.

All of the debugging features described in the [Gdb] reference manual are available for remote programs, but note:

1) While you may be able download a program via Ethernet, or some other high-speed mechanism, you will usually still need some other connection (e.g. EJTAG or serial cable) by which *gdb* can control the monitor. No known MIPS boards support a complete download and debug cycle over Ethernet alone.

2) Once a program has started running it cannot be restarted simply by using the *gdb*'s `run` command – the initialised data has most likely been modified by the program, and must be reinitialised by reloading the program first. The Insight GUI can do this for you automatically when you press the ''Run'' button.

Please refer to the printed or online GDB manual for more information about the GDB command line interface. See Chapter 13 ''Insight Graphical Debugger'' for a brief description of the graphical interface.

14.1 MDI Debugging

MIPS Technologies promotes a software API called ‘‘MDI’’; it’s a standard procedural interface by which host software can connect to an EJTAG probe or software simulator, via a dynamically loaded library conforming to the *Microprocessor Debug Interface* (MDI) specification.

Once you have configured MDI for the first time, following the instructions below, it is as easy to operate as any other *gdb* remote target. A typical command-line debug session might start like this:

Host System

```
$ mdi mipssim3[1]
$ sde-gdb xxxram[2]
(gdb) b main[3]
(gdb) target mdi 8[4]
(gdb) load[5]
(gdb) run[6]
Breakpoint 1 at main...
```

If you are using the Insight GUI it’s even simpler. Just click on the ‘‘Run’’ button (the *running man* icon), and when the Target Selection dialog appears for the first time select the ‘‘MDI Connection’’ target and the CPU device type. These selections are saved automatically when you exit Insight.

The following sections look in more detail at setting up and using the two most common MDI targets: the MIPSsim simulator and an MDI-enabled EJTAG probe.

14.1.1 MDI Debugging with the MIPSsim™ Simulator

MIPS Technologies Inc has developed the comprehensive and accurate MIPSsim simulator for its core CPUs. It is supplied with SDE as part of the MIPS® Software Toolkit bundle. It is not available for SDE *lite* users, who must use the GNU simulator, see Section 14.3 ‘‘Debugging with the GNU Simulator’’. The MIPSsim software runs on Windows (NT, 2000 and XP), x86 Linux, and Solaris 2.6 or above.

14.1.1.1 Configuring the MIPSsim™ Simulator for GDB

Sde−gdb connects to the MIPSsim simulator via its MDI library interface, and there are a few configuration steps which you must perform first, so that *gdb* can ‘‘find’’ the MIPSsim library.

We recommend that you install the MIPSsim package before installing SDE, so that the SDE install script can automate this configuration process for you. But if you didn’t do this, or if you later install a MIPSsim update into a new directory, then you will need to set this up manually, as follows:

1) First install, configure and test your MIPSsim package, following the instructions in the MIPSsim *Getting Started Guide* supplied with it.

2) *Sde−gdb* finds the MDI library using environment variables. Since you may need to switch between different MDI libraries (e.g. different MIPSsim versions, or between the simulator and an EJTAG probe), SDE includes a command-line tool called *mdi* which maintain these variables for you. It is controlled by small shell script ‘‘fragments’’ which tell it which environment variables have to be changed for each MDI library. To create a new MIPSsim MDI fragment simply enter the following command:

   ```
   $ mdi new mipssim
   ```

   You will then be asked for:

   a. A short, memorable name to give to this configuration, e.g ‘‘mipssim3’’ or ‘‘default’’. If you use the name ‘‘default’’ then this MDI configuration will be selected automatically by the SDE startup scripts when you login or open a new shell window – you won’t need to use the explicit *mdi* command shown below.

   b. A longer descriptive name for the configuration.

   c. The name of the directory where you installed the MIPSsim package – the same as the `MIPSARCHROOTn` setting described in the MIPSsim *Getting Started Guide*. In fact if the `$MIPSARCHROOT` variable is already

set, then you will be offered the chance to inherit that setting.

From now on you can select your new MIPSsim configuration using the *mdi* command followed by the short configuration name, for example:

```
$ mdi mipssim3
$ sde-gdb helloram
(gdb) target mdi 15:1
```

To see a list of all available configurations, simply enter:

```
$ mdi avail
  none          - Select this to clean your MDI environment
* mipssim3      - MIPSsim v3.4.14
  mipssim4      - MIPSsim v4.0.17
  fs2           - FS2 EJTAG probe
```

Note how the currently selected configuration is indicated by an initial ‘‘*’’.

The environment variables set up by the *mdi* command will be inherited by any sub-shells or other programs which you start from the same window. But they will not be remembered across sessions or between windows – apart from the ‘‘default’’ configuration, which is loaded automatically, you will have to reselect the chosen configuration each time you log in.

Tip: If you have received MIPSsim releases for more than one CPU core then, so long as they have the same MIPSsim version number, you can install them all into the same directory – simply let the common files overwrite each other. This will allow you to select between cores using GDB, or Insight’s GUI interface, rather than having to use an explicit *mdi* command before starting the debugger.

14.1.1.2 Selecting the MIPSsim™ CPU

When you connect to the MIPSsim simulator you have tell it which CPU core to simulate. You do this by specifying an MDI *target group* and *device* pair. The way that you do this depends on whether you are using the command-line or GUI interface to *gdb*.

1. For the command-line interface to *gdb* enter these commands:

```
$ sde-gdb
(gdb) show mdi devices
Targ 01: Default
    Dev 01: MIPS32_4Kc BE
    Dev 02: MIPS32_4Kc LE
    Dev 03: MIPS32_4Km BE
    Dev 04: MIPS32_4Km LE
    Dev 05: MIPS32_4Kp BE
    Dev 06: MIPS32_4Kp LE
    Dev 07: MIPS32_4KEc BE
    Dev 08: MIPS32_4KEc LE
    Dev 09: MIPS32_4KEm BE
    Dev 10: MIPS32_4KEm LE
    ...
```

   That should print out a list of all the CPU *devices* supported by the MIPSsim software, and their associated target group and device numbers. If it instead says ‘‘MDI not available’’, then you have probably not installed the MIPSsim package correctly, or not run the *mdi* command to select the MIPSsim library.

   Now you can tell *gdb* which device to use. Assuming that you wanted a little-endian 4KEc core, then looking at the above list we can see that it’s target group 1, device 8. So:

   a. Set the `GDBMDITARGET` and `GDBMDIDEVICE` environment variables to the appropriate target group and device numbers.

For bash, ksh, etc:

```
export GDBMDITARGET=1
export GDBMDIDEVICE=8
```

For csh and tcsh:

```
setenv GDBMDITARGET 1
setenv GDBMDIDEVICE 8
```

b. Or add the following gdb commands to to your `.gdbinit` file:

```
set mdi target 1
set mdi device 8
```

c. Or specify them on the `target` command line when you connect to the MDI library, for example:

```
$ sde-gdb
(gdb) target mdi 1:8
```

2. The Insight GUI interface is simpler – you select the MIPSsim CPU type interactively. Start *sde−insight*, open the *File->Target Settings...* menu or press the "Run" button, and set the *Target* field to "MDI Connection". Now pick the MDI CPU name from the list in the *Device* dropdown. These settings will be stored automatically in the `.gdbtkinit` (UNIX) or `gdbtk.ini` (Windows) preferences file in your home directory. If the *Device* field does not drop down a list of CPUs, then you have probably not installed the MIPSsim package correctly, or not run the *mdi* command to select the library.

Note that in both cases MIPSsim's MDI interface currently lists all of the CPU cores which it knows about, even if that core simulator is not installed. If you select a CPU type for which you do not have the core simulator library installed, then you will see an error reported when you try to connect to it.

14.1.1.3 Building for a MIPSsim™ Target

Use Table 8-1 "Supported target boards and simulators" to select an appropriate value of `SBD` which most closely matches your chosen CPU family, with the "`MSIM`" prefix. Now you can build one or more of the SDE example programs and run them on the MIPSsim simulator, for example:

1) Change directory to the "hello world" example program:

```
$ cd .../sde/examples/hello
```

2) Build the example:

```
$ sde-make SBD=MSIM32L
```

3) You can run the program in command-line mode:

```
$ sde-gdb helloram
(gdb) target mdi
(gdb) load
(gdb) run
...
(gdb) quit
```

4) Try running the program using the *Insight* graphical interface:

i) Start *gdb* with the command "**`sde-insight helloram`**"

ii) The main Insight *Source Window* will open. If the *Console Window* doesn't also appear, then click on the "console" icon in the source window's toolbar. This allows you to see output messages from the program being debugged.

iii) Click the "Run" icon (the running man) in the source window toolbar – the *Target Connection* dialog box will appear. Select "MDI Connection" in the *Target* field of the dialog box, then select your CPU type in the *Device* field, and click "OK".

iv) The program will be "downloaded" into the simulated MIPSsim RAM, then run until it hits a breakpoint in `main()`.

v) Click the "Continue" button (→{}) on the toolbar. The program will print "Hello World!" in the console window, and then stop at the next breakpoint, in the `exit()` function.

vi) Select "Exit" from the source window's "File" menu.

14.1.1.4 Downloading to a MIPSsim™ ROM Target

If you use the supplied example Makefiles then you can probably skip this section. We include it in case you need to write your own Makefiles, or in case something goes wrong.

When you build a program to blow into a physical ROM memory (e.g. EPROM or Flash) the SDE Makefiles will normally use the *sde−conv* program to convert it into an ASCII S-record file (or similar), suitable for a PROM programmer. At the same time its initialised, writable data segment is relocated and concatenated to the end of the code segment, from where it is later copied down into RAM. But *gdb* can't load an S-record file, so how do you load a ROM image into a bare MIPSsim simulator via *gdb*?

The answer is that *sde-conv* takes your executable ELF file, and outputs a new, *relocated* ELF file with the "`.relf`" extension. The relocation is done exactly the same way as when creating a real, physical PROM image.

The final step in the chain is that *gdb*'s "load" command automatically checks for a file with the same name as your executable, but with the "`.relf`" extension. If this is found then it is this file that will actually be downloaded via MDI into the simulated MIPSsim ROM. When execution is started the ROM startup code will (after initialising caches, etc) copy the initialised data and possibly code into RAM. Your program image will now correspond to the original ELF executable file, and debugging can begin.

Finally, if you are not using *gdb* to load and run the program, but wish to load a program directly into the MIPSsim simulated ROM using the `APP_FILE` setting in the MIPSsim configuration file, then remember to use the "`.relf`" file, not the original ELF file.

14.1.1.5 Non-standard MIPSsim™ Configurations

By default GDB will dynamically create a MIPSsim CPU configuration file to match your selected CPU type. It does this from a template stored in file `.../share/mipssim.cfg`. While this will a sufficient MIPSsim configuration to get you going, if you later need to change any of the CPU or memory parameters, or add new device or CorExtend libraries, then you'll need to create your own MIPSsim configuration file.

You can do this using either the MIPSsim GUI, supplied as part of the MIPSsim package, or by using a simple text editor. Full details of the configuration file format are contained in the MIPSsim documentation. The crucial configuration settings which you must change from the defaults supplied with the MIPSsim package are as follows:

Table 14-1 MIPSsim™ Configuration Settings

`APP_FILE`	Must be blank, or commented out.
`DUMP_FILE`	Must be blank, or commented out.
`BIG_ENDIAN`	To avoid a warning message set this to match your program's endianness.
`TRACE_FILE`	In v4.x of the MIPSsim simulator, just setting this will cause a trace log to be written to that file. You may not want to do that for normal debugging, since it will slow down the simulator.

You then have to tell *gdb* and the MIPSsim library how to find the configuration file which you just created, either:

a. Set the GDBMIPSSIMCONFIG environment variable to the name of the file, e.g.

For bash, ksh, etc:

```
export GDBMIPSSIMCONFIG=/path/to/myconfig.cfg
```

For csh and tcsh:

```
setenv GDBMIPSSIMCONFIG /path/to/myconfig.cfg
```

b. Or set it in the local .gdbinit file as follows:

```
set mdi configfile /path/to/myconfig.cfg
```

c. When using the Insight GUI, open the "Target Selection" dialog, select the "MDI Connection" target, and then enter the file name into the "Config" field.

It may be that you are happy to use GDB's default CPU core configuration file, but want to define a new device configuration file with more realistic memory timings, or new device models. GDB will add a reference to your device configuration file to its auto-generated core configuration file if you do one of the following:

a. Set the GDBMIPSSIMDEVCFG environment variable to the name of the device configuration file.

b. Or set it in the local .gdbinit file:

```
set mdi devcfgfile /path/to/mydev.cfg
```

14.1.2 MDI Debugging with an EJTAG Probe

MIPS Technologies is encouraging EJTAG probe manufacturers to offer an MDI interface to their devices. This provides a powerful way to debug system software using *gdb* at the lowest level, directly controlling the CPU core.

14.1.2.1 Configuring your probe for GDB

1) First follow the installation instructions supplied with your probe hardware, and check that you can access and control your CPU core via the probe vendor's own command-line debug tool. You'll need to make sure that the directory containing the probe's MDI DLL has been added to your PATH variable, or copy the DLL to \windows\system on Win9x, or \winnt\system32 on WinNT and above.

2) Use the *mdi new* command described in Section 14.1.1 "MDI Debugging with the MIPSsim™ Simulator" to create an MDI library configuration "fragment" for your probe's MDI library. There are two possible scenarios here:

   a. To create an FS2 EJTAG probe configuration, first install the FS2 software if it hasn't been already, and then enter this command:

```
$ mdi new fs2
```

   You will be asked for a memorable short configuration name, and a long description, to be displayed by the *mdi* command. The setup script will automatically search for the FS2 library in your PATH and LD_LIBRARY_PATH environment variables.

   b. For other probe vendors, whose installation tool has already added the directory containing their MDI DLL to the PATH or LD_LIBRARY_PATH environment variable, enter this command:

```
$ mdi new cutdown
```

   In addition to asking you for a memorable configuration name, and a long description, you will be asked to enter the name of the MDI DLL (e.g. something like "xxxmdi.dll" on Windows, or "libxxxmdi.so" on Linux).

   b. If the probe vendor's installation tool did not already change the PATH or LDPATH environment variable to include the directory containing their DLLs, or if you want to override the directory, then instead enter:

```
$ mdi new generic
```

This will ask you to enter additional information, including the name of the directory containing the probe DLLs.

4) Now you can select your probe configuration and run *sde−gdb*, for example:

```
$ mdi fs2
$ sde-gdb helloram
(gdb) target mdi
```

14.1.2.2 Selecting the EJTAG CPU

EJTAG probes connected by USB or parallel port probably support only one CPU at a time – the one to which it is currently connected. In that case you can probably connect to the probe without having to specify an MDI device number. But with some probes you may have to tell their MDI interface the name of the CPU, or the probe's Ethernet address, or some such. This selection can be made following exactly the same procedure described for selecting a MIPSsim CPU type in Section 14.1.1.2 "Selecting the MIPSsim™ CPU".

14.1.2.3 Building for an EJTAG-connected Target

Use Table 8-1 "Supported target boards and simulators" to select an appropriate value of `SBD` which most closely matches your chosen CPU family and evaluation board. This will have either the "`MALTA`" or "`SEAD`" prefix, but crucially it will have the "`J`" suffix, which indicates that the run-time system is configured to peform console and file i/o via MDI, rather than using the YAMON i/o system.

Now you can build one or more of the SDE example programs and run them on your target board, for example:

1) Change directory to the "hello world" example program:

```
$ cd .../sde/examples/hello
```

2) Build the example:

```
$ sde-make SBD=MALTA32LJ
```

3) You can run the program in command-line mode:

```
$ sde-gdb helloram
(gdb) set mdi connectreset 7
(gdb) target mdi
(gdb) load
(gdb) run
...
(gdb) quit
```

4) Try running the program using the *Insight* graphical interface:

   i) Start *gdb* with the command "**`sde-insight helloram`**"

   ii) The main Insight *Source Window* will open. If the *Console Window* doesn't also appear, then click on the "console" icon in the source window's toolbar. This allows you to see output messages from the program being debugged.

   iii) Click the "Run" icon (the running man) in the source window toolbar – the *Target Connection* dialog box will appear. Select "MDI Connection" in the *Target* field of the dialog box, then select your CPU type in the *Device* field, and click "OK".

   iv) The program will be "downloaded" to the board, then run until it hits a breakpoint in `main()`.

   v) Click the "Continue" button (→{}) on the toolbar. The program will print "Hello World!" in the console window, and then stop at the next breakpoint, in the `exit()` function.

   vi) Select "Exit" from the source window's "File" menu.

14.1.2.4 Resetting the CPU

When you connect to a remote CPU via an EJTAG probe to download and run your program, you may want to simultaneously reset the CPU to ensure that it always starts in a known good state. However on many evaluation boards the reset signal will also reset the memory controller, which will prevent you (and *gdb*) from accessing DRAM until it has been programmed.

Rather than teaching *gdb* how to initialise your memory controller, the simplest thing to do is allow the onboard PROM monitor (e.g. the YAMON monitor) to run just long enough to program the memory controller, and then halt the CPU so that *gdb* can take control. This behaviour is controlled by *gdb*'s "`mdi connectreset`" setting, which can have the following values:

- *Off*: is the default value, and in this case *gdb* does not try to reset the the remote CPU, it simply halts it. Note that for this to work you may need to modify your probe software's configuration files to prevent *it* from automatically resetting the CPU.
- *On*: In this case *gdb* will reset the CPU and then halt it immediately. You shouldn't use this unless your memory controller automatically resets into a usable state, or you are willing to use gdb commands to program it manually.
- *N*: In this case *gdb* will reset the CPU, allow it to run for *N* seconds, and then halt it. For the MIPS Malta board the value 7 is usually suffcient to allow the YAMON monitor to initialise the board.

You can effect this setting in a number of different ways:

1) Set it in the local `.gdbinit` file as follows:

```
set mdi connectreset 7
set mdi connectreset on
```

2) Or set the `GDBMDICONNRESET` environment variable:

For bash, ksh, etc:

```
export GDBMDICONNRESET=7
export GDBMDICONNRESET=0            # On
```

For csh and tcsh:

```
setenv GDBMDICONNRESET 7
setenv GDBMDICONNRESET 0
```

3) Or you can set it each time you enter the "target" connect command, by appending "`,rst=N`" to the device number. For example:

```
(gdb) target mdi 1,rst=7
```

4) Or, when you use the Insight GUI, the "Target Settings" dialog box – which appears when you first hit the "Run" button – has a "Reset on Connect" tickbox option which enables the reset, and a field in which to enter the number of seconds to pause after the reset.

14.1.3 MDI Debugging Tips

14.1.3.1 Command line arguments

If your application has been linked with the standard SDE run-time system, then you can pass command-line arguments to your application (via *argc* and *argv*) when debugging via MDI:

1) When using the *gdb* command-line interface, append the arguments to *gdb*'s ''run'' command, or set the *gdb* ''args'' variable. See the [Gdb] reference manual for more details.
2) When using the Insight GUI interface you can put your arguments in the ''Arguments'' field of the ''Target Selection'' dialog, when you click on the ''Run'' button.

14.1.3.2 MDI Host File I/O

If your application has been linked with the standard SDE run-time i/o system, then console and file i/o requests will be passed via the MDI interface to *gdb*. You can see your program's output in *gdb*'s console window. If your program attempts to read from its console, then you can input text through *gdb*'s console window when you see the ''`app>`'' prompt. Your program can also read and write files on your host computer – see Section 19.1.1.1 ''Host File Access'' for more details.

Beware that this i/o mechanism will not work if you don't use *gdb* to load and run your program; for example if you load the program directly into MIPSsim using the `APP_FILE` setting in the MIPSsim configuration file. In such cases you must find some other way to perform console and file i/o, such as via an additional MIPSsim device which you provide.

14.1.3.3 MDI Variables and Commands

The MDI interface adds a number of new *gdb* variables and commands which provide finer grain control over the MDI library and its attached CPU than would normally be available with remote *gdb* targets.

`set mdi stepinto`
: When set to on an MDI single-step will always execute exactly one instruction – if an interrupt or exception occurs then execution will stop with the PC pointing to the start of the exception handler. In environments where interrupts are occurring faster than the time it takes to step through the interrupt handler, it may not be possible to make any progress in the foreground application in this mode.

  When off, a single-step will always execute one instruction in the foreground application, ignoring asynchronous interrupts. This may be implemented simply by disabling interrupts globally while single-stepping.

  The variable defaults to off.

`set mdi threadstepall`
: Selects simultaneous TC stepping mode when scheduler locking is enabled. When on, all TCs are stepped together, otherwise single-stepping only enables execution in the selected TC. Defaults to off.

`set mdi continueonclose`
: When set, the target will be told to restart CPU execution when *gdb* closes its MDI connection. If off, then the target will be reset when the connection is closed. Defaults to on.

`set mdi rununcached`
: If on then the program's start address is forced to an uncached address, since it may need to initialise the caches before trying to execute code. When false the start address is not changed. Defaults to on.

`set mdi waittime`
: Sets the number of milliseconds which MDI should wait before returning a result to *gdb*, when waiting for the run/halt state of the CPU to change. Some MDI libraries ignore this. It defaults to 10ms.

`set mdi library` *`NAME`*
: The name of the MDI DLL to connect to. Initialised to the value of the `GDBMDILIB` environment variable, if available.

`set mdi configfile` *`NAME`*
: The name of the MIPSsim CPU configuration file. Initialised to the value of the `GDBMIPSSIMCONFIG` environment variable, if available. See Section 14.1.1.5 "Non-standard MIPSsim™ Configurations".

`set mdi devcfgfile` *`NAME`*
: The name of the MIPSsim device configuration file. Initialised to the value of the `GDBMIPSSIMDEVCFG` environment variable, if available. See Section 14.1.1.5 "Non-standard MIPSsim™ Configurations".

`set mdi target` *`TARGNUM`*
: The MDI *target group* number to connect to. Defaults to the value of the `GDBMDITARGET` environment variable, if available.

`set mdi device` *`DEVNUM`*
: The MDI *device* number to connect to. Defaults to the value of the `GDBMDIDEVICE` environment variable, if available.

`show mdi devices`
: Displays a list of the available MDI target groups and devices. The MDI DLL library name must be known before this will work.

`set mdi prompt`
: Sets the prompt to use when the application program requests console input. Defaults to "`app>`".

`set mdi asid auto|off|on|`*`ASID`*
: Controls which address space to use when accessing mapped virtual addresses through the TLB, and for qualifying breakpoints. When set to "`off`" it uses the global address space; when "`on`" it uses the current ASID value in the CPU's `EntryHi` register; when "`auto`" it uses the global address space for unmapped address, and the current ASID for mapped addresses; otherwise it must be an explicit numeric ASID (0 to 255). Defaults to "`auto`". Breakpoints use the same setting to qualify the breakpoint request, which on certain targets may allow breakpoints to be triggered only when executed by a specific ASID.

`show mdi tlb [`*`INDEX`*`]`
: Displays the contents of the TLB. *`INDEX`* is an optional TLB index, else the whole TLB is displayed.

`set mdi tlb` *`INDEX HI LO0 LO1 MASK`*
: Programs the *`INDEX`*'th entry in the TLB using the values *`HI`*, *`LO0`*, *`LO1`* and *`MASK`*.

`show mdi cp0` *`REG`*`[/`*`BANK`*`]`
: Displays arbitrary Coprocessor 0 registers which are not normally accessible via *gdb*. The argument *`REG`* is the register number; */`BANK`* is the optional bank number, default 0.

`set mdi cp0` *`REG`*`[/`*`BANK`*`]` *`VALUE`*
: Sets arbitrary Coprocessor 0 registers which are not normally accessible via *gdb*.

`show mdi icache|dcache|scache` *`ADDRESS [, SET]`*
: Displays the contents of one line in the CPU's primary instruction, primary data or secondary cache. The *`ADDRESS`* argument is a byte offset into the cache, and *`SET`* is the cache set. Note that *`SET`* is optional, and if present a comma is required as separator between the two arguments; if absent then all sets at that cache offset are displayed.

  This command has the side-effect of setting *gdb* internal variables `$ctag`, `$cparity`, `$cdata0`, `$cdata1`, etc to the values displayed. If multiple sets are displayed, then only the highest numbered set is recorded in these variables.

`set mdi icache|dcache|scache` *`ADDRESS, SET, TAG, PARITY, DATA, ...`*
: Sets the contents of one line in the CPU's primary instruction, primary data or secondary cache, using the values provided. Note that a comma is required as separator between the values.

`set mdi connectreset on|off|`*`N`*
: See Section 14.1.2.4 "Resetting the CPU".

`set mdi gmonfile` *`NAME`*
: Sets the file name to which *gdb* will write *gprof* profiling data, when enabled. The default file name is "`gmon.out`".

`set mdi connecttimout N`
The number of seconds for which *gdb* will wait for a target to halt execution when first connecting to it. The default is 1 second, set to 0 for unlimited timeout. GDB may be safely interrupted while it is waiting for the halt to complete.

`set mdi gmonfile` *NAME*
Sets the file name to which *gdb* will write *gprof* profiling data, when enabled. The default file name is "`gmon.out`".

`set mdi profile`
If set to "on", and you are using the MIPSsim simulator, then *gdb* will tell the simulator to collect profiling information which gdb will write to *gmonfile* when the program exits. If set to "auto", then *gdb* will automatically collect and output the profiling data, but only if your program contains the `_mcount` symbol, which will be the case if your program was compiled with profiling enabled. The default is "auto".

`set mdi profile-cycles`
If set then, if MIPSsim profiling is enabled, *gdb* will tell the simulator to count cycles rather than instructions. This will only work of your MIPSsim software is licensed for cycle counting. Defaults to off. This can also enabled using the "`mdi cycles enable`" command, described below. In MIPSsim 4.0 and above you select whether you want cycle counting or not by the MDI device which you connect to – this setting will have no effect.

`set mdi profile-mcount`
If set then *gdb* includes the `_mcount` function in the profile data. Defaults to off, which doesn't profile `_mcount`.

`set mdi mcount-symbols` *SYM ...*
A list of symbol names in the executable which may label the function which is used to collect call-graph profile data, and should be excluded from the profile data unless `mdi profile-mcount` is set. Defaults to "`_mcount __mcount`".

`set mdi ftext-symbols` *SYM ...*
A list of symbol names in the executable which may define the start of the executable code segment, for profiling. Defaults to "`_ftext`".

`set mdi etext-symbols` *SYM ...*
A list of symbol names in the executable which may define the end of the executable code segment, for profiling. Defaults to "`_ecode _etext`".

`set mdi logfile` *NAME*
Name of a file in which to store a trace of calls made to the MDI library, for troubleshooting. Requires that GDB's `debug remote` is set to 1 or 2. It must be set before issuing the `target mdi` command.

`mdi cacheflush`
Causes dirty lines in the CPU data cache to be written to memory, and then invalidates all CPU caches.

`mdi cycles enable`
Enable MIPSsim cycle counting, if licensed. From this point on *gdb*'s `$cycles` convenience variable will be set to the current cycle count. By using the command `display $cycles` you can then see how many cycles have been used as you step through your code. In MIPSsim 4.0 and above you select whether you want cycle counting or not by the MDI device which you connect to – this command has no effect.

Also in MIPSsim 4.0 the counter includes the cycles required to flush the pipeline when an MDI breakpoint or single-step causes execution to stop, and to restart the pipeline when resuming execution. So there will be an overhead per breakpoint or step command which you will need to subtract.

`mdi cycles clear`
Clears the MIPSsim cycle counter to zero, and then enables cycle counting. The Insight GUI runs this command if you click on the "clapperboard" icon in the Source window.

`mdi cycles disable`
Disables MIPSsim cycle counting. Has no effect with MIPSsim 4.0 and above.

`mdi cycles status`
Reports on whether MIPSsim cycle counting is available, and if so whether it is enabled or diabled.

`mdi reset [WHAT]`
It may sometimes be useful to start over from the reset vector when debugging system firmware. The optional argument can be one of the following:

`full`
Reset the entire target system, if possible. This is the default action if no argument is given, and is often the only action supported by the hardware. The CPU will exit the reset state and halt before fetching the first instruction from the memory location at the reset vector.

`device`
If the device consists of a CPU plus peripherals, reset both if possible.

`periph`
If the device consists of a CPU plus peripherals, reset just the peripherals if possible.

`cpu`
If the device consists of a CPU plus peripherals, reset just the CPU if possible.

`mdi regsync`
Forces *gdb* to write back any modified register values to the target CPU. Normally this only occurs when *gdb* is about to restart execution of the application.

`monitor COMMAND...`
Sends the command line to the MDI library's "do command" interface. The command line is not interpreted by *gdb*.

14.1.3.4 MDI troubleshooting

If your MDI-connected probe or simulator appears to be misbehaving then it will help us to help you if you collect a log file which shows the MDI calls which occur between GDB and the MDI library. You may be able to work out what's going wrong for yourself, by looking at this file, but if not then please send it to us along with a log of your GDB session.

You can create a log file by switching on remote debug mode **before** issuing the *target* command, and then repeating whatever commands cause your problem, e.g.

```
(gdb) set mdi logfile mdilog.txt
(gdb) set debug remote 2
(gdb) target mdi
...
(gdb) quit
```

14.2 Debugging with MIPS® MT ASE

To better understand the rest of this chapter, it will help if we first describe a couple of the fundamental terms defined in the MIPS MT (multi-threading) ASE:

- *Thread Context (TC)* : The hardware state necessary to support a single thread of execution within a multi-threaded CPU device. This includes a set of general purpose registers, multiplier registers, a program counter (PC) and a small amount of privileged state.
- *VPE* : A virtual processing element (VPE) is an instantiation of the full privileged CPU state on a multi-threaded CPU, sufficient to run an independent per-processor OS image – it can be thought of as a virtual CPU. Each VPE must have at least one TC bound to it in order to execute instructions and be debuggable, but it may contain more than one TC when running an explicitly multi-threaded OS or application. A conventional single-threaded CPU could be thought of as implementing a a single TC bound to a single VPE.

These components of the MT ASE may be used within a variety of programming models, with different debugging methodologies:

- *LLMT* : Low-Level Multi-Threading (LLMT) describes programs which make explicit use of the hardware TCs to run multi-threaded code, but generally with no more software threads than there are hardware TCs. This may be a self-contained threaded application, or a simple RTOS kernel. When debugging such software you will want to track the behaviour of the hardware TC states as they execute threads within your program. See Section 14.2.1 “Debugging LLMT Applications” below.

  Be aware that LLMT debugging only provides visibility of threads which are are assigned to hardware TCs. Neither the probe, simulator, nor GDB have the OS-specific knowledge to find and interpret a stored thread context in target memory. So if you are debugging a threaded application which has more threads than TCs to run them, presumably with a micro-kernel or RTOS to context switch the threads between TCs, then you will need to use the debugging facilities or thread-aware remote debugging protocol provided by the OS to debug application-level threads. The LLMT model may however be used to bring-up and debug the kernel.
- *AP/RP* : The term AP/RP describes a programming model where the VPEs are treated as independent loosely coupled cores, with one acting as “Application Processor” (AP), running a complex operating system such as Linux; while the other acts as the “Real-time Processor” (RP), running dedicated real-time code without interference from the AP operating system’s scheduling and interrupt handling. This mechanism has sometimes been known as AP/SP and AMVP.

  Debugging an application program on the AP side use the standard OS application debugger, but debugging a program running on the RP side requires the **sde−gdb** debugger, with something like a remote serial or EJTAG probe connection to the RP VPE, as described in Section 14.2.3 “Debugging AP/RP Applications” below.
- *SMVP* : Describes the execution of a largely unmodified symettric multi-processing (SMP) operating system by multiple VPEs. In this environment application programs – including multi-threaded applications – will be debugged using the OS’s usual debugger, and the underlying MT hardware will typically be “invisible” to the application programmer. See Section 14.2.4 “Debugging SMVP/SMTC Programs” below.
- *SMTC* : An extension of the SMVP model which requires more significant modifications to an SMP operating system, so that it can schedule multiple software threads and/or processes to run on the hardware TCs. As with SMVP, the OS’s normal application debugger will typically be used for debugging threaded applications; for kernel debugging the LLMT model may be applicable.

14.2.1 Debugging LLMT Applications

When you debug low-level or operating system kernel code which makes explicit use of hardware TCs (the “LLMT” model described above) you can use **sde−gdb** in conjunction with an MDI library that supports the multi-threading extensions. At the time of writing this means a recent version of 34K MIPSsim or the FS2 EJTAG probe. The following section assumes that you have succesfully connected GDB to your target via a suitable MDI library, as described in Section 14.1 “MDI Debugging”.

When you connect **sde−gdb** to a MT-capable CPU via an MT-aware MDI library, then the hardware TCs can be accessed using GDB’s thread debugging facilities – for full details of these commands see the “Debugging programs with multiple threads” section of the GNU GDB manual (supplied as HTML and PDF with SDE). To

illustrate how these facilities map onto hardware TCs, the critical features are also documented below.

14.2.1.1 Thread Status

Whenever execution stops and control returns to GDB, the debugger will display which TCs have been activated or deactivated since the last prompt and, if it has changed, the name of the current thread. One thread is always the ‘‘thread of interest’’ to which all GDB commands will apply by default, and this is the ‘‘current thread’’. When GDB first regains control from the application the current thread will be the TC which hit the breakpoint, or completed a single-step. In the case of asychronous stop (e.g. pressing the Insight Stop button, or typing Ctrl-C in the command line GDB) then any TC may be chosen as the current thread.

You can display a list of all active TCs, and their program counters within the program, as follows:

```
(gdb) info thread
   2 Thread Context 4 in client_thread()
 * 3 Thread Context 2 in server_thread()
   ...
```

Note that there are two numbers on each line: first GDB’s thread number, and secondly the hardware Thread Context (TC) number. All of the GDB thread commands work in terms of GDB thread numbers (the first number), not the hardware TC numbers (the second number).

So this particular example tells you that GDB’s thread number 2 corresponds to hardware TC 4, and its program counter is within the `client_thread()` function; while GDB’s thread number 3 corresponds to TC 2, and its program counter is within the `server_thread()` function. Thread 3 (TC 2) is marked with an asterisk to indicate that it is the current thread.

If you are using the Insight GUI then you can open the ‘‘Thread List’’ window to view the active TCs and switch between them. Selecting a different TC will change the source window, stack window and register window to reflect the state of the selected TC.

14.2.1.2 TC-specific breakpoints

You can set an breakpoint that will only ‘‘trigger’’ when executed by a specific TC simply by appending the `thread` qualifier to a breakpoint command. For example:

```
(gdb) b send_message thread 2
```

This will set a breakpoint in the `send_message` function to be activated only when executed by ‘‘client’’ thread (TC 4) listed above. You can also set thread-specific breakpoints using the Insight GUI by right-clicking on the left of a line in the source window.

Beware that a software breakpoint exception will be taken by every TC which executes the breakpoint instruction, requiring communication between GDB and the target. GDB will quietly step over breakpoints which occur in the wrong TC, but performance will be substantially reduced.

It is not currently possible to specify a TC-specific hardware data watchpoint. A hardware watchpoint set up using GDB’s ‘`watch`’, ‘`rwatch`’ or ‘`awatch`’ commands will trigger when any TC bound to the same VPE accesses that location in the specified manner.

14.2.1.3 Thread-specific commands

You can switch GDB from one TC to another using the ‘`thread`’ command, or its alias ‘`t`’ e.g.

```
(gdb) thread 2                  # switch to "client" thread
(gdb) bt                        # do stack trace of "client"
(gdb) t 3                       # switch to "server" thread
(gdb) info reg                  # display registers of "server"
```

Alternatively you can perform the same operation on a number of threads at once, e.g.

```
(gdb) thread apply 1 2 7 4 bt    # apply backtrace cmd to threads 1,2,7,4
(gdb) thread apply 2-7 9 p foo   # apply p foo cmd to threads 2->7 & 9
(gdb) t apply all x/i $pc        # apply x/i $pc cmd to all threads
```

14.2.1.4 Resuming threaded execution

Normally when you issue a single-step command there is no guarantee which TCs will run in which order – they might even hit a breakpoint before your single-step request completes, ‘‘seizing the prompt’’ away from your original thread of interest. A sequence of single-step commands may switch you back and forth between your active TCs, or just advance the highest priority one.

To avoid this happening while single stepping you may disable execution of all other TCs on the same VPE, apart from the currently selected TC, by using this command:

```
(gdb) set scheduler-locking step
```

But beware that this can get you into situations where the TC which you are stepping cannot make any progress, because it is waiting for a semaphore or mutex to be unlocked by another TC – so it is not always the most appropriate behaviour.

Furthermore this command:

```
(gdb) set scheduler-locking on
```

will prevent other TCs on the same VPE from running in all cases, even when you resume execution using commands like ‘`continue`’, ‘`until`’ or ‘`finish`’.

Finally, rather than locking out the other TCs altogether, you can request that all TCs should ‘‘gang step’’ together. This requires both GDB ‘`scheduler-locking`’ and ‘`mdi threadstepall`’ to be set. For example

```
(gdb) set scheduler-locking on
(gdb) set mdi threadstepall on
```

In summary:

scheduler-locking	mdi threadstepall	Single-step Behavior
`off`	×	Current TC single-steps, all other TCs run freely until the current TC completes an instruction, or one of the other TCs hits a breakpoint
`on|step`	`off`	Current TC single-steps, all other TCs are suspended
`on|step`	`on`	All TCs single-step together – the first to complete an instruction returns GDB to command mode, selected as the current thread

14.2.2 Debugging Multiple VPEs

Debugging multiple VPEs within a multi-threaded core is very similar to debugging multiple independent CPUs within a multi-core system.

14.2.2.1 Multiple VPEs with FS2 probe

The rest of this section assumes that you have installed, configured and selected EJTAG probe software as your current MDI target, as described in Section 14.1.2 ‘‘MDI Debugging with an EJTAG Probe’’. For reliable multi-VPE debugging it is recommended that you use version 2.1.6.7 or higher of the FS2 probe software – contact `support@fs2.com` for details of the recommended versions.

When you start GDB with the probe connected to a 34Kc™ core, you should see something like this in response to the ‘`show mdi devices`’ command:

```
(gdb) show mdi devices
Targ 01: mips-single-core
    Dev 01: mips-single-core-root
Targ 02: mips-dual-cores
    Dev 01: mips-dual-cores-mips1
    Dev 02: mips-dual-cores-mips2
Targ 03: mips-34k
    Dev 01: mips-34k-vpe1
    Dev 02: mips-34k-vpe0
```

If that works as described, then you should now be able to connect to VPE0. Assuming the same numbering as above then use the '`target mdi 3:2`' command, which should result in output as follows (the last line, especially the address reported, will vary):

```
(gdb) target mdi 3:2
Selected device mips-34k-vpe0 on MIPS unknown
[New Thread Context 0]
Connected to MDI target
0x8010049c in ?? ()
```

You are now set up and ready for debugging.

General VPE debugging with probe

To access a VPE within an multi-threaded CPU the appropriate target group number must be used (e.g. target group 3 in the example above). Within that group the device number 1 corresponds to VPE1 and the device number 2 corresponds to VPE0. Thus to attach to VPE0 you need to use '`target mdi`' and then select group number 3 and then device number 2 interactively, or alternatively use '`target mdi 3:2`'. Likewise for VPE1 you could use '`target mdi 3:1`'. If you are using the Insight GUI then the devices can also be selected from the "Target Connection" dialog.

A given VPE can only be usefully connected to if it has at least one thread context (TC) bound to it. Therefore with the default configuration, VPE0 can be controlled straight from RESET, but VPE1 can only be once some code has been run to bind a TC to it. However, GDB may be attached to a disabled VPE and it will keep waiting until it has been activated.

GDB may be used to load a program to be debugged to the target. A typical session in this case is going to include the following commands in the given order:

```
$ sde-gdb program
    ... start GDB and load program's symbol table
(gdb) target mdi 3:2,rst=7
    ... connect to VPE0, and reinitialise the target
(gdb) load
    ... transfer the program to the target's memory
(gdb) break function
    ... set a software breakpoint on a function
(gdb) run
    ... start execution
```

Note the '`rst=7`' option when connecting to VPE0. That tells GDB to reset the target CPU just after establishing the connection. Execution is then resumed and the target is allowed to run freely for seven seconds, after which it is halted. The intent is to let the firmware (e.g. YAMON) initialize board resources, in particular the caches and memory controller, so that the target can accept a program image. Note that this will reset the whole board and CPU, not just the selected VPE, so it would not make sense to use this option when connecting to VPE1.

Depending on your setup, to load a large program into memory you may either use the probe via the GDB '`load`' command, or it may be faster to use the system's firmware. For the latter, and a board like the Malta, that would be YAMON (see the YAMON manual for how to do this); for other systems it would be system-specific. Sometimes you may be may be debugging the firmware itself. An example session may look like this:

```
$ sde-gdb program
    ... start GDB and load program's symbol table
(gdb) target mdi 3:2
    ... connect to VPE0 - VPE0 is halted
(gdb) break function
    ... set a software breakpoint on a function
(gdb) continue
    ... resume execution of already loaded program/firmware
```

If it's the firmware being debugged, it may sometimes be useful to start over from the reset vector. For this, the '`mdi reset`' command may be useful – this resets the target system entirely. The CPU will exit the reset state and stop before fetching the first instruction from the memory location at the reset vector. You normally really want to issue this command from a debugger connected to VPE0 as VPE1 will become inactive as a result.

When a debugging session is terminated, the VPE can either be left halted or execution may be resumed, therefore letting code that has been previously debugged run freely. Use the following commands to control that behaviour:

```
(gdb) set mdi continueonclose on
    ... to resume execution
(gdb) set mdi continueonclose off
    ... to keep the target halted
(gdb) show mdi continueonclose
    ... to retrieve the current setting
```

14.2.2.2 Multiple VPEs on the MIPSsim™ simulator

The rest of this section assumes that you have installed, configured and selected the MIPSsim simulator as your current MDI target, as described in Section 14.1.1 "MDI Debugging with the MIPSsim™ Simulator". For reliable multi-VPE debugging it is recommended that you use version 4.6.36 or higher of the MIPSsim software – contact `support@mips.com` for details of the latest versions.

MIPSsim provides two ways of working with multiple VPEs. One uses a single MDI device to access the whole core: all thread contexts are accessible through a single connection to the device, regardless of the VPE to which they are bound. The other way uses a pair of separate MDI devices where each has access only to thread contexts bound to the corresponding VPE. The second method requires an auxiliary program called **mipssimd** that controls internal communication between the two MDI devices – this tool is supplied as part of the MIPSsim package, but is not currently available for Windows hosts.

Setting up mipssimd

Working with **mipssimd** requires additional settings to be present in environment variables. They are necessary for the program to create identifiable communication channels with clients connecting to VPE 0 and VPE 1 of the same simulated processor. System V IPC is used. The variables are as follows:

`$MIPS_MDI_IPC_KEY`
: defines a file to be used as a key to identify this particular instance of a simulated processor. The file has to exist and be accessible. This variable is also used by GDB to select which instance of **mipssimd** to communicate with.

`$MIPS_MDI_IPC_CLIENTS`
: defines the number of clients to be handled. For the 34K family this has to be set to "2" for the 2 VPEs the processor implements.

`$MIPS_MDI_IPC_CLIENT_ID`
: defines the number of the communication channel to use between the debugger and a single instance of **mipssimd**, starting from '0'. For the 34K this can be either '0' or '1'. This variable is only used by GDB, and each instance of GDB should have a different value.

With `$MIPS_MDI_IPC_KEY` and `$MIPS_MDI_IPC_CLIENTS` set up you should be able to start **mipssimd**. But before that, it's generally a good idea to clean up any leftover state in IPC resources that may have been left from previous **mipssimd** runs. There is a dedicated program included with MIPSsim that does that. To run it, enter the '`mdiipcwatchdog cleanup`' command. You should get output like below (obviously the path to the key

file will differ, depending on the value of $MIPS_MDI_IPC_KEY, as may the key and the seed). The following example assumes a Bourne-style shell, for a C shell use the 'setenv' command.

```
$ touch /home/joe/.MIPS_MDI_IPC_KEY
$ export MIPS_MDI_IPC_KEY=/home/joe/.MIPS_MDI_IPC_KEY
$ export MIPS_MDI_IPC_CLIENTS=2
$ export MIPS_MDI_IPC_CLIENT_ID=0
$ mdiipcwatchdog cleanup
Destroying shared memory and semaphores.
Generated key '0x1157340' using key string
´/home/joe/.MIPS_MDI_IPC_KEY' and key seed 0x1
```

This cleanup step is not required before running **mipssimd** for the first time, but as a side effect it also validates the setup, so running it anyway is a sensible idea.

Now to actually run **mipssimd**, you should see output as follows (again, the path to the key file will likely differ):

```
$ mipssimd -p -f
Starting up......

Establishing connection with debuggers using key
´/home/joe/.MIPS_MDI_IPC_KEY'.
Support up to 2 debugging clients
Ready to handle IPC commands from debugger #0.
Ready to handle IPC commands from debugger #1.
```

If this is works, then you are ready to start working with **mipssimd**.

The options given to **mipssimd** above have the following meaning:

`-p` stands for ''persistent'' and makes **mipssimd** preserve the state of the simulated system between MDI connections

`-f` stands for ''forever'' and makes **mipssimd** keep running even when the last client disconnects.

The result is to make **mipssimd** behave like a real h/w CPU, allowing multiple debugger connections to be opened and closed, until it is terminated. Once you finish debugging, you may terminate **mipssimd** by sending it the SIGINT signal. It's done in a system-specific way, usually by typing <Ctrl>+<C>, also written as `^C` – run '`stty -a`' and see the entry marked '`intr=`' for what character is used in a given system – or by using the shell's '`kill`' command. With the latter, bear in mind **mipssimd** is multithreaded and all threads must be terminated.

General VPE debugging with simulator

MIPSsim provides two target group numbers for the 34K – number 21 is for the instruction-accurate simulator and number 22 is for the cycle-counting version. Within each of the groups six devices are defined as follows:

1	Whole CPU, little-endian
2	Whole CPU, big-endian
3	VPE0, little-endian
4	VPE0, big-endian
5	VPE1, little-endian
6	VPE1, big-endian

Thus to attach to VPE0 of a big-endian, cycle-counting 34K you need to use the '`target mdi`' command and select the group number 22 and then the device number 4 interactively or alternatively use '`target mdi 22:4`'. Similarly for a whole CPU access to a little-endian, instruction-accurate 34K you may either select the group number 21 and then the device number 1 or use '`target mdi 21:1`'.

GDB may be used to load a program to be debugged to the target. A typical session in this case is going to include the following commands in the given order:

```
$ sde-gdb program
    ... start GDB and load program's symbol table
(gdb) target mdi 21:2
    ... connect to whole 34K, instruction accurate, big-endian
(gdb) load
    ... transfer the program to the target's memory
(gdb) break function
    ... set a software breakpoint on a function
(gdb) run
    ... start execution
```

Normally GDB generates a MIPSsim configuration file on the fly from a template (installed as `.../share/mipssim.cfg`) and uses this whenever a target is opened. If the default settings are unsuitable, then a custom configuration file may be used. Once such a file has been created, use the following command in GDB to use it instead of the default auto-generated file:

```
(gdb) set mdi configfile filename
```

Sometimes it's useful to start debugging a program that has already been loaded into MIPSsim memory; this can done using the `APP_FILE` setting in a MIPSsim configuration file. An example session may then look like this:

```
$ sde-gdb program
    ... start GDB and load program's symbol table
(gdb) set mdi configfile myconfigfile
    ... select custom configuration file
(gdb) target mdi 22:1
    ... connect to whole 34K, cycle-accurate, little-endian
    ... the simulator loads the application executable file
    ... the device is halted
(gdb) break function
    ... set a software breakpoint on a function
(gdb) continue
    ... resume execution under control of GDB
```

14.2.3 Debugging AP/RP Applications

The mechanism for debugging a program running on the RP side of an AP/RP system is similar to downloading and running a "bare-iron" program on a target board connected by a serial port or network. It is also possible to debug RP programs using an EJTAG probe.

14.2.3.1 Using the SP Debugging Daemon

This mechanism allows debugging of an RP program without use of a h/w EJTAG probe. The remote debug connection is via TCP/IP, with the GDB remote debug protocol transported between **sde-gdb** on the development host, through a network server running on the target CPU's AP Linux VPE, and then via a shared memory FIFO to the RP VPE.

In the current implementation the debug protocol is finally interpreted by a remote debug "stub" which is linked into your RP application, similar to the remote serial debugging of standalone programs described in Section 14.4.2 "Serial Debugging with SDE Debug Stub".

The following example demonstrates how to debug an RP application running on a Malta board. Debugging an RP application on MIPSsim is not currently possible using this mechanism.

1) Build your application following the Section 9.1.3 "Command Line Monitor (minimon)" example, but with the **RDEBUG** makefile variable set to `imm`, e.g.

```
devhost$ cd .../sde/examples/minimon
devhost$ sde-make clean
devhost$ sde-make all RDEBUG=imm SBD=MALTA32LSP
```

2) If you haven't already done so, then start the SP debugging daemon on your Malta Linux target:

```
aplinux$ spd &
```

3) Open a new connection to you Malta (e.g. using **telnet**', **ssh**', **rlogin**', etc) in another terminal window, and start the AP/RP **rtterm**' (real-time terminal) application. This will allow you to communicate with the running RP program's virtual console:

```
aplinux/2$ rtterm
```

4) Now transfer the `minirel` program to your Malta board and"download" it into the Signal Processor by sending it to the `/dev/vpe1` device on the Application Processor Linux host, e.g. in your original window:

```
aplinux$ cat minirel >/dev/vpe1
```

5) Now start **sde–gdb** (on your development host) and connect it via TCP/IP to the debug server running on the target board:

```
devhost$ sde-gdb minirel
(gdb) target remote aplinux:2222
```

In the above *aplinux* represents the network hostname or IP address of your Malta board. GDB automatically determines the load address of the relocatable program, and relocates its symbol table data to match.

6) Now you can set breakpoints and enter the GDB '`c`' or '`continue`' command to start the program running under the control of GDB. Don't use the '`r`' or '`run`' command to start execution, since this would restart the RP program from its entrypoint.

7) Note that the `spd` remote debugging daemon does not currently support interrupt requests from GDB, so it is not possible to break into a runaway RP application from GDB by typing Control-C or pressing the Insight Stop button. To diagnose such problems you will need to use other techniques such as breakpoints to find the problem; or use an EJTAG probe, which can interrupt any program, whatever its state.

14.2.3.2 AP/RP Debugging with EJTAG Probe

Refer to Section 14.2.2 "Debugging Multiple VPEs" for general information on using a probe to debug multiple VPEs.

The usual method of debugging an AP/RP Linux kernel with an EJTAG probe is to let the firmware load and start Linux and then attach to VPE0 (the Linux AP) which is already running. Similarly for VPE1 (the RP program), except that the Linux VPE loader is used to load and start the program.

Since the probe firmware does not know the load address of a relocatable RP program, and cannot tell GDB how to relocate its symbol table, it's usually easier to debug a fully linked RP executable (i.e. an executable called `*ram` rather than `*rel`). To build such a program, assuming that you have started your Linux kernel with the '`memsize=30M`' boot option, you would build your program something like this:

```
devhost$ cd .../sde/examples/minimon
devhost$ sde-make clean
devhost$ sde-make ram SBD=MALTA32LSP DLBASE_C=81e00000
```

Note that `81e00000` is the KSEG0 (untranslated, cacheable) mapping of the Linux maximum memory size (30MB = `0x1e00000`). Also note that the RDEBUG option should not be used when debugging using an EJTAG probe.

If debugging of either AP or RP from the very beginning of the loaded program is required, then hardware execution breakpoints may be placed at the entry point. Use the GDB's '`hbreak`' for this. It accepts any syntax that is valid for the '`break`' command; in particular absolute numeric addresses may be specified after an asterisk. As the command uses a hardware breakpoint register in the debug port of the core it has to be issued to the correct VPE and will not affect the other VPE.

If the RP-side VPE to be debugged is inactive, then there is no way to set a hardware breakpoint since on an attempt to connect GDB will stall, waiting for the VPE to become activated. GDB will pass control to the user as soon as the VPE becomes active and before the first instruction of the program has been executed.

14.2.3.2.1 AP/RP Team Debugging

Sometimes when doing debugging it may be desired for the VPEs involved to be stopped and resumed synchronously, so that the state of the target system remains as stable as possible during debug accesses. GDB provides a way of doing that by grouping VPEs, and potentially any devices, into the so called teams. While a single instance of GDB can only fully control one device at a time, including the device in a team with other devices makes requests for stopping and resuming be propagated to all of them. If any of the other devices have instances of GDB attached to them, these requests are transparent to their controlling debuggers. Specifically a device in a team that has been stopped by another debugger, but not the controlling one, stops, but continues reporting the running state to the latter. Likewise a device that has been resumed by the controlling debugger starts reporting the running state, but resumes only after all the other debuggers resumed it.

The following commands are used to control teams:

`target mdi <device>,team=<device>[,team=<device>...]`
: Open the device specified at the beginning and attach it together with ones listed as '`team=`' arguments to the currently selected team.

`mdi team attach <device> [<device>...]`
: Attach listed devices to the currently selected team.

`mdi team detach <device> [<device>...]`
: Detach listed devices from the currently selected team.

`mdi team clear`
: Destroy the currently selected team removing all members beforehand, the currently selected team is set to "0".

`mdi team list`
: List identifiers and members of the currently existing teams.

`set mdi team <id>`
: Select a team identifier for further team operations, "0" means a new team will be created for attachment operations.

`mdi team <id>`
: Shorthand for '`set mdi team <id>`'.

`show mdi team`
: Print the identifier of the currently selected team.

`mdi team`
: Shorthand for "show mdi team" .

  The use of these commands is incompatible with group debugging as described in Section 14.2.4.2 "SMVP/SMTC using FS2 Probe and Group Debugging".

14.2.3.3 AP/RP Debugging with MIPSsim

Note that this feature is not supported by the current release of the AP/RP package for TimeSys Linux.

One way to debug such a setup is to use a custom MIPSsim configuration file to load and run the AP/RP Linux kernel. It can be used straight from GDB using the '`set mdi configfile`' command. In such a setup after opening the target, programs as referred to from the configuration files will have been loaded into MIPSsim memory and may be started just by issuing the '`continue`' command. Soon you will see Linux kernel messages being output. Depending on whether **mipssimd** is used or not, they will appear through **mipssimd** or GDB's window. This communication channel is actually the Linux console and once the user mode is reached will accept input as well.

Similarly with VPE1 (RP), the Linux VPE loader is the usual way of starting the program, rather than loading it through the MDI interface. It's usually easier to use a fully linked executable, as described above for the EJTAG probe. Memory space for loading such an executable has to be reserved in the MIPSsim device configuration file – one provided with the Linux AP/RP package may be used as the starting point (see the MIPSsim documentation for anything that is not immediately obvious in the file).

Since the simulator returns control to GDB after loading Linux, the kernel may also be debugged from the very beginning as is – rather than issuing '`continue`' you may use any commands, like '`step`' or '`break`' to set up debugging as required.

If debugging of the RP program from the very start of the loaded program is required, then a hardware execution breakpoint may be placed at the entry point. Use the '`hbreak`' command of GDB for that. If split per-VPE devices and **mipssimd** are used, then '`hbreak`' has to be issued to the correct VPE and it will not affect the other one. A connection to the VPE1 device has to be made and the breakpoint be set within, which will trigger as soon as VPE1 executes the instruction there.

14.2.4 Debugging SMVP/SMTC Programs

14.2.4.1 SMVP/SMTC using MIPSsim® Simulator

On MIPSsim the use of the ''whole CPU'' device to debug shared program image operating systems running across multiple VPEs is recommended – refer to Section 14.2.2.2 ''Multiple VPEs on the MIPSsim™ simulator'' above. In this case TCs bound to any VPE all become visible and controllable as threads within GDB, as described in Section 14.2.1 ''Debugging LLMT Applications'' above. All active VPEs also halt and resume execution simultaneously. This is probably what you would expect anyway, when debugging SMP operating systems on a single 34K.

14.2.4.2 SMVP/SMTC using FS2 Probe and Group Debugging

Group debugging allows synchronous control of multiple devices by a single instance of GDB. All thread contexts of all open devices are seen as threads of a single running program. This is most useful for debugging SMP-style execution environments, though it is not strictly required for each of the devices to excecute the same code. Internally the devices are synchronised to one another, that is, events causing one device to stop freeze all the other ones and if GDB decides to return control to the user, then all the threads have their state preserved as of the time of the event, subject to hardware or simulator limitations.

The FS2 MDI libraries prior to version 2.1.8.0 do not fully support device synchronization. When using them, GDB still permits doing group debugging, but there is no synchronisation between devices and the state preserved will only be a rough approximation of what the system would look like if a debugger was not attached. This may still be useful for debugging systems which have no strict timing restrictions.

Use the following command to debug a group of devices:

`target mdi <device>,group=<device>[,group=<device>...]`

Open all the devices requested at once. The use of this command is incompatible with team debugging as described in Section 14.2.3.2.1 ''AP/RP Team Debugging''.

Here is an example session for SMTC Linux:

```
(gdb) file ./vmlinux
Reading symbols from /home/macro/linux/vmlinux...done.
(gdb) target mdi 2:2,group=2:1,rst=0
Selected device mips-dual-cores-mips2 on MIPS unknown
Selected device mips-dual-cores-mips1 on MIPS unknown
[New Thread Context 2:2:0]
Connected to MDI target
0xbfc00000 in ?? ()
(gdb) continue
Continuing.
```

[Here Linux is started from YAMON.]

```
^C
Quit received: Stopping target
[New Thread Context 2:2:0]
[New Thread Context 2:2:1]
[New Thread Context 2:2:2]
[New Thread Context 2:1:3]
[New Thread Context 2:1:4]

Program received signal SIGINT, Interrupt.
[Switching to Thread Context 2:1:3]
0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
48              __asm__(".set\tmips3\n\t"
(gdb) info threads
  5 Thread Context 2:1:4  0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
* 4 Thread Context 2:1:3  0x80101e6c in r4k_wait () at arch/mips/kernel/cpu-probe.c:48
  3 Thread Context 2:2:2  r4k_wait () at arch/mips/kernel/cpu-probe.c:48
  2 Thread Context 2:2:1  r4k_wait () at arch/mips/kernel/cpu-probe.c:48
  1 Thread Context 2:2:0  0x8035a5f4 in _spin_unlock_irqrestore (lock=0x80407774,
    flags=1024) at kernel/spinlock.c:284
```

Notice how device numbers are reported prefixing thread context numbers above.

14.3 Debugging with the GNU Simulator

You can debug a program using the GNU MIPS simulator which is built into *sde−gdb*. It works very like any other remote debug mechanisms – in fact internally it looks to gdb like a remote board.

As supplied the GNU simulator does not simulate i/o devices[18], just a bare MIPS architecture CPU, RAM and a set of PROM monitor entrypoints. So you can't use the GNU simulator to run programs built for a real hardware target like a Malta board – you must build your programs specifically for the GNU simulator target, e.g. `SBD=GSIM32B`.

For a more complete example of building and debugging a program using the GNU Simulator see Chapter 5 "Quick Start".

You can see your program's output in *gdb*'s console window. If your program attempts to read from its console, then you can input text through *gdb*'s console window when you see the '`app>`' prompt. Your program can also read and write files on your host computer – see Section 19.1.1.1 "Host File Access" for more details.

[18] Actually, if you are brave, then it is possible to add device models to the GNU simulator by editing the source.

14.4 Remote Serial Port Debugging

If you've got a MIPS Technologies evaluation board such as the Malta or SEAD-2 boards, but you haven't got an EJTAG probe, then you'll probably be debugging your programs using a remote debug protocol over the serial port. You also might need to use serial debugging in other cases, such as when you need to debug a multi-threaded application or RTOS, which requires a debug protocol that can handle software thread contexts – for example MDI can provide access to low-level hardware TCs on a multi-threaded CPU (see Section 14.2 "Debugging with MIPS® MT ASE "), but does not know how to find or interpret the state of a software thread which is not currently assigned to a hardware TC.

GDB serial ports

When you connect to a target using a serial (RS232) port, you have to tell *gdb* the name of the port device to use. In the examples which follow we've chosen to use the Linux device name `/dev/ttyS0`, but this is operating system specific, and you'll have to use different names as appropriate for you host. Table 14-2 "Host O/S serial port devices" gives a list of possible names for different operating systems.

Table 14-2 Host O/S serial port devices

Host	Device names
Linux	`/dev/ttyS0, /dev/ttyS1`
Windows	`/dev/com1, /dev/com2`
Solaris	`/dev/ttya, /dev/ttyb`

GDB serial protocols

There are several different ways that a MIPS program can be debugged remotely, and the distinction often causes confusion.

1) Using the default gdb serial remote debug protocol, support for which is built into the YAMON monitor on MIPS Technologies boards, or

2) Again using the default gdb serial remote debug protocol, but in this case connecting to the SDE remote debug stub, which can be linked into your program if you are building a rommable or "standalone" program, or

3) Using the historical MIPS Computer Systems remote debug protocol, as implemented in some PROM monitors (e.g. *IDT/sim* and *PMON*). But this mechanism is no longer documented in this manual. It is a completely different debug protocol, and requires *different commands* to get it started.

The amount of data passed back and forth between the board and *gdb* means that some operations can be quite slow at 38400 baud (the YAMON monitor's default speed). You can use *sde−gdb*'s **−b** option, its '`set remotebaud`' command, or the Target Selection dialog in the GUI, to raise the serial line speed to 57600 or 115200 baud, if the target board can handle it. Where the host/target link is slow it's quicker to set *gdb* temporary breakpoints (the '`tbreak`' command) and then '`continue`', rather than doing repeated '`step`' commands. You can also speed things up by enabling *gdb*'s memory transfer cache using the '`set remotecache`' command, but don't do that if you plan to use *gdb* to access device registers or shared memory.

14.4.1 Serial Debugging with the YAMON™ Monitor

The YAMON PROM monitor supplied on MIPS Technologies' Atlas, Malta and SEAD-2 boards implements *gdb*'s default remote debug protocol. The YAMON gdb protocol is "hardwired" to use the board's second serial port (*tty1*), so you will usually need two serial connections between the host and the board: one connected to a terminal emulator for the console, and one used by gdb for the remote debug protocol.

The YAMON monitor runs its serial ports at a default 38400 baud, and in some cases (slow FPGA-based cores) may require hardware flow-control to avoid UART receive buffer overruns. This can be enabled by *gdb*'s `set remoteflow` command, or using the h/w flow control tickbox in the GUI's "File−>Target Settings..." dialog.

14.4.1.1 YAMON™ Monitor – Serial Download

Follow this example to load a program `xxxram` over a serial port to a board running the YAMON monitor (e.g. built with `SBD=MALTA32L`).

Target Console

```
YAMON> gdb[5]
```

Host System

```
$ sde-gdb xxxram[1]
(gdb) set remotebaud 38400[2]
(gdb) set remoteflow on[3]
(gdb) b main[4]

(gdb) target remote /dev/ttyS0[6]
(gdb) load[7]
(gdb) cont[8]
```

14.4.1.2 YAMON™ Monitor – TFTP Download

If you have an Ethernet connection to your board and a TFTP server on your host, then you can avoid a long serial download by downloading your program over Ethernet with the YAMON monitor's `load` command, and then starting *gdb* as follows:

Target Console

```
YAMON> load tftp://192.168.1.1/xxxram.s3[5]
YAMON> gdb[6]
```

Host System

```
$ sde-gdb xxxram[1]
(gdb) set remotebaud 38400[2]
(gdb) set remoteflow on[3]
(gdb) b main[4]


(gdb) target remote /dev/ttyS0[7]
(gdb) cont[8]
```

To simplify this further you could set the YAMON *$start* environment variable to run the YAMON `load` and `gdb` commands after every reset.

14.4.1.3 YAMON™ Monitor via Insight – Serial Download

Using the Insight GUI with the YAMON monitor is slightly more tricky than when using the MIPSsim or GNU simulators:

1) Using your terminal emulator, issue the *gdb* command via the YAMON console, e.g.

```
YAMON> gdb
```

2) Start *gdb* on your host, with the GUI interface.

```
$ sde-insight xxxram
```

3) Click on the running man icon to bring up the ‘‘Target Settings’’ dialog: select the ‘‘Remote/Serial’’ target; select the host serial port which is connected to the YAMON debug port; select a baud rate of 38400 baud.
4) Still in the ‘‘Target Settings’’ dialog, click on ‘‘More Options’’ and make sure that ‘‘Attach to Target’’, ‘‘Download Program’’ and ‘‘Continue from Last Stop’’ are all ticked.
5) Press the OK button and your program will download (slowly, over the serial port) and run.
6) When the program terminates you have to go right back to step (1) to reload it again.

14.4.1.4 YAMON™ Monitor via Insight – TFTP Download

If you want to use TFTP loading over Ethernet, then follow these steps:

1) In your terminal emulator download your program using the YAMON *load* command, e.g.

```
YAMON> load tftp://192.168.1.1/xxxram.s3
```

2) Issue the YAMON *gdb* command:

```
YAMON> gdb
```

3) Start *gdb* on your host, with the GUI interface.

```
$ sde-insight xxxram
```

4) Click on the running man icon to bring up the ‘‘Target Settings’’ dialog: select the ‘‘Remote/Serial’’ target; select the host serial port which is connected to the YAMON debug port; select a baud rate of 38400 baud.
5) Still in the ‘‘Target Settings’’ dialog, click on ‘‘More Options’’ and make sure that ‘‘Attach to Target’’ and ‘‘Continue from Last Stop’’ are both ticked, but ‘‘Download Program’’ is not.
6) Press the OK button: *gdb* should connect to the YAMON monitor and start running your program.
7) When your program terminates you have to go right back to step (1) to reload it again. You could set the YAMON *$start* variable to run the YAMON `load` and `gdb` command after every reset.

14.4.2 Serial Debugging with SDE Debug Stub

The SDE run-time system includes a ‘‘remote debug stub’’, which implements the target monitor for *gdb*’s default remote debug protocol. This stub will only be linked into your application if the target board’s PROM monitor does NOT include one of the supported remote debug protocols, or if you are building a standalone, rommable or Signal Processor program. In both cases you must also define the **`RDEBUG`** *makefile* variable in the example makefiles, see Section 9.2 ‘‘Example Makefiles’’.

N.B. The **`RDEBUG`** variable is ignored when you build a program for a monitor which already supports *gdb* remote debugging. For example MIPS Technologies’ YAMON monitor also uses the gdb default remote debug protocol, but you should be reading the previous section, which describes YAMON debugging.

Before starting *sde−gdb* you have to start your program running. For a RAM-based program this will mean downloading it to your board, using whatever facilities your board’s monitor provides, and issuing some sort of ‘‘go’’ command. For a rommable program this might mean blowing an EPROM or Flash, plugging it into your board, and just switching it on!

Your program will now run until it gets an unexpected exception, at which point it displays a message on its console to indicate that it is waiting for the remote debugger to make contact. On your host system you can now start *sde−gdb* and perform post-mortem diagnosis as follows:

Target Console

```
<start program>[1]
SDE General Exception, reason=...
Cause 00000008
etc.
Awaiting remote debugger...
```

Host System

```
$ sde-gdb xxxrom[2]
(gdb) target remote /dev/ttyS0[3]
(gdb) bt[4]
```

If you want to set breakpoints *before* the program starts running, then define **`RDEBUG=immed`** when building it. The startup code will then stop and wait for *sde−gdb* just before entering your `main()` function. At this point you can connect *sde−gdb*, as above, set your breakpoints and continue. For example:

Target Console

```
<start program>[1]
Awaiting remote debugger...
```

Host System

```
$ sde-gdb xxxrom[2]
(gdb) target remote /dev/ttyS0[3]
(gdb) b main[4]
(gdb) c[5]
```

Note that most SDE board kits do not support serial port interrupts, so it is not usually possible to interrupt a runaway application from GDB when using the remote debug protocol, e.g. by typing Control-C, or pressing the Insight Stop button. To debug such problems you must use other techniques such as breakpoints to find the problem; or use an EJTAG probe which can interrupt any running program, even when interrupts are disabled.

If you wish to use a faster baud rate, then you will need to recompile the board-specific serial-port driver (i.e. `.../sde/kit/`*`SBD`*`/sbdser.sx`) with a larger value of the **DBGSPEED** constant defined (e.g. in the board's `sbd.mk` or `sbd.h` file). To run the debug protocol down the console port (i.e. sharing a single connection) define **DBGPORT=0** in `sbd.mk`; on boards which support a non-volatile environment the same effect can be achieved by setting either the *$dbgport* or *$hostport* board variable to "`tty0`".

14.4.3 Serial Comms Fault Finding

If your target board is not quite capable of keeping up with the data rate from the host (which can happen if your UART doesn't have a FIFO), or if some error is occurring in the remote debug protocol code, then *sde-gdb* may run very slowly, or mysteriously time-out the connection. If this happens then you should try switching on serial port logging in gdb before issuing the *target* command, and then repeat whatever commands cause the problem, e.g.

```
(gdb) set remotelogfile log.txt
```

When you close the target connection, the named file will contain a trace of all data sent and received by *gdb*.

You can also try

```
(gdb) set debug remote 1
```

which tells the higher-level remote protocol code to output debug information about its activity.

With the YAMON monitor you can ask the remote end to output a debug protocol log to the console, by starting it up with the **–v** flag, like this:

```
YAMON> gdb -v
```

The debug trace information is naturally somewhat cryptic if you are not familiar with the protocols, but you may be able to identify dropped characters or other problems. If you need to contact us with a debug comms problem, then it will be helpful if you can email the trace information to us.

14.5 Debugging C++

Works as advertised in the GDB manual, so long as you use the default DWARF-2 debug format. The alternative "Stabs" format used in previous releases of SDE can also be used, but is deprecated. The DWARF-1 format does not support C++.

Profiling with GPROF and GCOV

Profiling allows you to learn where your program spent its time and which functions called which other functions while it was executing. This information can show you which pieces of your program are slower than you expected, and might be candidates for rewriting to make your program execute faster.

Profiling is very useful because human programmers seem to guess very badly about which parts of a program take the CPU most effort; even for small programs the results may surprise you.

Sometimes, profiling can also help trace bugs, by telling you which functions are being called more or less often than you expected. A *code coverage* report allows you to check that all parts of your application have been exercised.

The profiler uses information collected during the actual execution of your program. However, how your program is run will affect the information that shows up in the profile data. If you don't use some feature of your program while it is being profiled, no profile information will be generated for that feature. Of course your program will also run much slower than normal, which may make it difficult to profile applications with critical real-time constraints.

Note that the collection of profiling data requires a significant amount of extra RAM on your target. In general you need at least as much free memory as the size of your code segment, twice as much if you don't have a remote file i/o facility with which to upload the data to your host.

15.1 Compiler Options for Profiling

Here is a summary of the compiler flags used to tell the compiler to instrument your code to collect profiling data, and the types of profiling which are supported by SDE – consult the [Gprof] reference manual for more details.

15.1.1 Statistical (PC-sampling)

This technique involves running your program and statistically sampling the value of the program counter using a regular clock interrupt (typically 100Hz). The PC sample histogram is written to a *gmon.out* file, which is read by *sde-gprof* to generate a *flat profile* – a simple sorted table showing you in which functions, statistically, your program has spent most of its time.

This doesn't require any special compiler flags or instrumentation of your code – it only requires that the C startup code calls `_gmoncontrol()` to start the sampling interrupt. However it is usually used in conjunction with call graph collection, as follows.

15.1.2 Function Call Graph

When you compile your program with the **-pg** option, the compiler inserts a call to the `_mcount()` function into each function prologue. This constructs a call graph: a data structure which records the dynamic function call history – which function called which others, and how often.

The function call graph is written to a *gmon.out* file, along with the PC sample histogram described above. *Sde-gprof* combines these to present a report which shows you not only where your code spent most of its time, but how it got there.

If you don't have access to all of the source code to compile with **-pg** – for example when using third party libraries – then *gprof* can fill in the gaps by building a static call graph. This is generated by walking the program and tracing static function calls. Functions which are discovered this way won't have dynamic call count information, and the static graph can't trace indirect calls, but it can help you interpret the data. You make this happen by giving *gprof* the **-c** option.

15.1.3 PC Counting

When you profile a program using the MIPSsim simulator, the profiling data is collected very differently from other targets. Instead of sampling the PC at intervals, the MIPSsim simulator can count *every instruction* or, if it is licensed to do so, *every cycle*. This permits a much more accurate analysis of the code's behaviour, and eliminates problems with sample aliasing. It can be collected both with or without the function call graph (i.e. with or without the additional compiler instrumentation, which can itself effect the behaviour of caches, etc).

15.1.4 Line Granularity

The *sde−gprof* program can generate a more fine-grained report attributing time to individual source lines, instead of complete functions. In order to do this it is only necessary to compile your program with line number debugging enabled (the **−g** flag).

15.1.5 Compiler Profile Feedback

If you compile your program with the **−fprofile−generate** option, the compiler will insert instrumentation code which records profile information of interest to the compiler. When fed back to the compiler with **−fprofile−use**, this data will improve its branch prediction, loop unrolling/peeling, basic block reordering, register allocation and so on − this is known as profile-directed optimization. See also Section 12.2.1 "Optimizing for Speed".

15.1.6 Code Coverage

If you use the **−ftest−coverage** and **−fprofile−arcs** options, the compiler will output data files which contain the flow-graph information required by the *sde-gcov* program to generate a code coverage report. The *gcov* program is documented in the [Gcc] reference manual.

15.2 Example Makefile PROFILE Option

The previous sections listed the different types of profiling available, and the compiler flags associated with them. To simplify the use of these tools with the SDE example programs, our makefile build system has a shorthand mechanism for building a program with various types of profiling enabled. Just define the `PROFILE` variable when you build the program − for example the following will completely rebuild the example benchmark with call-graph and pc-sample profiling enabled:

```
$ cd .../sde/examples/dhrystone
$ sde-make SBD=MALTA32LJ PROFILE=yes clean all
```

Note the use of the "`clean`" and "`all`" targets, used together to delete and then rebuild the whole application using the new compiler options in a single step.

Here's the complete list of `PROFILE` values available:

PROFILE=	Compiler Flags	Description
no		No profiling (the default)
yes	`-pg`	Instrument code for call graph and pc sampling
line	`-pg -g`	Enable line granularity
feedback-generate	`-fprofile-generate`	Instrument for compiler feedback
feedback-use	`-fprofile-use`	Enable profile directed optimization
gcov	`-fprofile-arcs` `-ftest-coverage`	Instrument for code coverage

15.3 Profiling with the MIPSsim™ Simulator

PC profiling with the MIPSsim simulator requires *sde−gdb* to control the simulator's instruction and cycle counting, and then convert the resulting data into a `gmon.out` file which can be read by *sde−gprof*. The typical flow is described here.

15.3.1 Instruction counting

When you use the SDE makefile system to build a program, you can arrange to collect a PC sample and a function call graph for use by *sde−gprof* simply by setting the `PROFILE` variable to "`yes`". For example:

```
$ cd .../sde/examples/dhrystone
$ sde-make SBD=MSIM32L PROFILE=yes
```

Then, when you run your program using *sde−gdb*, as described in Section 14.1.1 "MDI Debugging with the MIPSsim™ Simulator", *gdb* will automatically collect the instruction count information and output it to file "`gmon.out`". The instrumentation code in your application will collect the call graph data, and output it using MDI host file i/o to the file "`mdi-gmon.out`". These two files are then merged by *sde−gprof* to generate the final report, as follows:

```
$ sde-gdb dhryram
(gdb) target mdi 15:1
(gdb) load
(gdb) run
... program runs
(gdb) quit
$ sde-gprof dhryram mdi-gmon.out gmon.out >profile.txt
```

See the [Gprof] reference manual for a detailed discussion of the *gprof* reports – but note that the results are now displayed not as seconds per function, but as dynamic instruction count per function.

Be aware that the instruction counts may be scaled: the scale factor will be reported at the beginning of the profile output. The MIPSsim software counts instructions using 32-bit counters, but the *gmon.out* format uses 16-bit counters, so *gdb* has to scale the data to fit. For measuring the relative impact of sections of code 16-bits is more than enough accuracy (do you really care about less than .001%?), but you should be aware of the potential loss of resolution on long profile runs.

15.3.2 Cycle counting

Now suppose that you want to count cycles rather than instructions, to see the effect of pipeline stalls caused by such things as cache misses, or instruction interlocks. If your MIPSsim software is licensed to allow cycle counting, then simply repeat the above process, but before you run your program enter the *gdb* command:

```
(gdb) set mdi profile-cycles
```

Now the `profile.txt` generated by *gprof* will have columns showing the number of cycles per function.

You may not want to use cycle counting all the time, because it will make simulator run much slower – instruction counting is sufficient in many cases.

In MIPSsim 4.0 and above you select whether you want cycle counting or not by the MDI device which you connect to – the "`profile-cycles`" setting has no effect.

15.3.3 Omitting the Call Graph

Let's suppose that you want a really accurate cycle based profile of your program. The trouble is that the instrumentation added by the compiler to collect the call graph will itself disrupt the performance of your program by polluting the caches with its own instructions and data.

First you must compile your program normally, i.e. don't set any of the profiling flags. You must then tell *gdb* to collect the MIPSsim profile manually; and finally run *gprof*, telling it not to expect or output a call graph. For example:

```
$ cd .../sde/examples/dhrystone
$ sde-make SBD=MSIM32L PROFILE=no
$ sde-gdb dhryram
(gdb) target mdi 8
(gdb) set mdi profile on
(gdb) set mdi profile-cycles
(gdb) load
(gdb) run
... program runs
(gdb) quit
$ sde-gprof -p dhryram gmon.out >profile.txt
```

15.3.4 Line Granularity

Compile your program using "`PROFILE=line`". Collect your MIPSsim profile data as normal. Then simply add the **–l** (letter ell) option when running *sde−gprof* to report the profile data with line granularity. For example:

```
$ sde-make SBD=MSIM32L PROFILE=line clean all
$ sde-gdb dhryram
...
$ sde-gprof -l dhryram gmon.out mdi-gmon.out >profile.txt
```

15.3.5 Interactive Cycle Counting

Another way to collect fine grain MIPSsim profile information is to do so interactively.

1) Load your program into *gdb* and set a breakpoint in the area that you want to examine. Run the program up to that point.

2) In the *gdb* console window enter this command:

```
(gdb) display $cycles
```

3) Then either enter the command:

```
(gdb) mdi cycles clear
```

3b) or if using the Insight GUI click on the "clapperboard" icon in the source window.

4) Now single step your program by source line or machine instruction. As you do so the accumulated cycle count will be displayed in the console window. You can reset the count at any time by repeating step (3).

5) Experienced *gdb* users could attach commands to breakpoints to control and collect the value of the `$cycles` variable.

15.4 Manual Instrumentation

A common technique used to measure accurately the performance of small sections of a program, is to manually insert code to sample the CPU's performance counter registers or real-time *Count* register, for example using the `mips_getcount`() function described in Section 20.6 "System Coprocessor (CP0) Intrinsics".

Take care when using this technique in combination with other tools such as the MIPSsim simulator and *gprof*, since it is is not easy to correlate the results. The GDB and MIPSsim profile data collection will not count cycles spent executing the gprof call graph code (i.e. the `_mcount` function), whereas these cycles will be included in the result obtained from `mips_getcount`(). Don't expect the results to match unless you omit the call graph instrumentation, following the instructions in Section 15.3.3 "Omitting the Call Graph" above.

15.5 Profiling with an EJTAG Probe

Profiling on a real CPU connected via an EJTAG probe is fully supported. However the profile information will be collected by the statistical PC sampling method, using a 100 Hz timer interrupt. The profile output files will be written to the host using the MDI host file i/o facility. In the case of the *gmon.out* data, the data will in fact be written to a file named "`mdi-gmon.out`", so the final step will be something like this:

```
$ sde-gprof dhryram mdi-gmon.out >profile.txt
```

15.6 Profiling with the YAMON™ Monitor

Profiling on a real CPU when running under the YAMON monitor is partially supported. But the YAMON monitpr has no remote file i/o interface by which the running software could access the profiling data files on the host. The arc profiling and code coverage facilities are therefore not supported, since they require real-time file access. But statistical PC-sampling and call graph collection are supported.

Since no file system is available, the run-time system just places the *gmon.out* file in memory and reports its address. It is then up to you to use whatever "upload" facilities your PROM monitor provides to transfer this region of

memory to a file on your host. For example:

```
YAMON> load
YAMON> go
Profiling data at 0X805656CC-0X8056E3AE (size 0x8ce2)
User application returned with code = 0x00000000
YAMON> fwrite tftp://192.168.238.25/gmon.out 805656cc 8ce2
About to binary write tftp://192.168.238.25/gmon.out
Successfully transferred 0x8ce2 (10'36066) bytes
YAMON>
```

Note that in the above example, most TFTP daemons will require that file `gmon.out` in the public TFTP directory must already exist and be publically writable, e.g.

```
host$ touch /tftpboot/gmon.out
host$ chmod a+rw /tftpboot/gmon.out
```

15.7 Profiling with the GNU Simulator

When running under the GNU simulator (*sde-run*) there is no clock interrupt with which to collect the PC sample data. Fortunately the PC sampling is performed internally by the simulator, which itself writes a *gmon.out* file containing the PC histogram. Using *gprof* you can then merge the PC-sample data in "`gmon.out`" with the call graph data collected by your instrumented application, which is written by the SDE run-time profiling code to file "`gsim-gmon.out`". Here's an example:

```
$ sde-make SBD=GSIM32L PROFILE=yes clean all
$ sde-run --profile-pc-granularity=4 dhryram
$ sde-gprof dhryram gmon.out gsim-gmon.out >profile.txt
```

Ignore the absolute execution times reported in the *gprof* output, since the GNU simulator is not cycle accurate and the sampling rate is based only on a simple instruction count. The execution time percentages are not cycle accurate: the simulator takes no account of cache misses, memory latency, instruction interlocks etc; nonetheless the data still gives you useful information about where your programs spends most of its time.

15.8 Profile-directed Optimization

This profiling technique does not require cycle accuracy, or any timing hardware: it is based solely on instrumenting your code to count the number of times each conditional branch in your program is taken, or not taken.

It does however need to run on a target which has access to the host file system. This means that it will run on a MIPSsim simulator, or a CPU connected via an MDI EJTAG probe, or the GNU simulator – but not on a target connected by a serial port (e.g. using the YAMON monitor).

1) Compile and link your program with "`PROFILE=feedback-generate`".
2) Delete any "`*.da`" files.
3) Download and execute your program to generate the arc count data files. When your program terminates, the profiling library will create a set of files named after your source files, but with the "`.da`" suffix. Each time you run your program the "`.da`" files are updated to merge in the new counts, so you can perform multiple runs with different data sets, to improve the coverage.
4) Compile and link your program with "`PROFILE=feedback-use`". which tells *gcc* to read the profiling data file and use it to direct its optimizations. For example:

```
$ sde-make SBD=MSIM32L PROFILE=feedback-use clean all
```

15.9 Code Coverage Report

This mechanism also needs a target which has access to the host file system, so it won't work on a YAMON target.

1) Compile and link your program with "`PROFILE=gcov`".

2) Delete any "`*.da`" files.

3) Download and execute your program to generate the profiling "`.da`" data files. Each time you run your program the "`.da`" files are updated to merge in the new data, so you can perform multiple runs with different data sets, to improve the coverage.

4) Run the *sde-gcov* tool to generate a code coverage report. For example:

```
$ sde-gcov foo.c
 87.50% of 8 source lines executed in file foo.c
Creating foo.c.gcov.
```

The file "`foo.c.gcov`" contains output from *gcov*. See the [Gcc] reference manual for more information on the *gcov* program.

Linker Scripts and Object Files

16.1 Linker Scripts

The linker (*sde−ld*) is always controlled by a script file, a default one is built into the linker. The default script combines all the input sections of the same name together, to form larger output sections, and it can be found in `.../lib/ldscripts/elf32mipssde.xn`.

You can copy and edit the script file to suit your particular requirements. The directory contains example scripts to link ECOFF and SGI-dialect ELF objects too, which you may find useful. The GNU Linker manual contains full details of the script language. Be warned: the script language is tricky, the language implementation somewhat fragile, and exotic use may well show up linker bugs. If you do anything other than use the ‘‘standard’’ scripts and small modifications, you should expect to work hard.

See Section 16.4 ‘‘Using Extra Sections’’ for an example of how linker scripts are used. We may already have a script that is suitable for your needs: contact us for details.

16.2 ELF Object File Format

SDE uses the ELF object file format, and aims to be able to interlink with most contemporary MIPS ELF versions. Reference information on MIPS ELF can be found in [ELF], [ABI] and [MIPSABI]. The format of the debug information passed from the compiler to the source-level debugger is independent; SDE currently prefers STABS.

ELF files can define multiple *sections*. Roughly speaking, the output of the assembler is a file containing one or more named sections; when two or more object files are linked, sections with the same name are combined; so the section ‘‘.text’’ is used for machine instructions, and by default all the instructions end up together. The compiler and the assembler generate quite a lot of different sections implicitly, and the default linker scripts built in to SDE know which *segment* (a segment is a chunk of the eventual program image) to put them in. See Table 16-1 ‘‘Standard ELF section names’’.

You can also deliberately place code or data in arbitrarily named sections if you want to take control over exactly where different chunks of your program end up in memory; see Section 16.4.3 ‘‘Linking Extra Sections’’ for how to do that.

Much of the time you won’t really be aware of all these sections, but when you use one of the binary utility programs in SDE – *sde−nm*, *sde−objdump*, *sde−readelf*, *sde−ld* and so on – you will see those names.

Table 16-1 Standard ELF section names

Section name	*What generates it*	*Where it ends up*
`.text`	Compiler- or assembler-generated instructions	Executable code segment
`.text.hot`	Functions which are called frequently	
`.text.unlikely`	Functions which are called rarely	
`.text.*`	Functions when compiled with **−ffunction−sections** are output to uniquely named sections of this form	
`.gnu.linkonce.t.*`	C++ methods – only one copy of each section with the same name is output to the code segment	
`.init`	Code to be run before *main* (e.g. C++ setup)	
`.fini`	Code to be run after *_exit* (e.g. C++ teardown)	

Section name	*What generates it*	*Where it ends up*
.rodata	Strings and C data declared `const`	Read-only data segment
.rodata.*	Constant data when compiled with **–fdata–sections** are output to uniquely named sections of this form	
.rodata.str*S.A*	Mergeable strings of size *S* and alignment *A*	
.rodata.cst*A*	Mergeable constant data of alignment *A*	
.gnu.linkonce.r.*	C++ “link-once” constant data	
.ctors	Pointers to C++ static constructors	
.dtors	Pointers to C++ static destructors	
.eh_frame_hdr	C++ exception handling information	
.eh_frame		
.gcc_except_table		
.data	Variables >*n* bytes (compiled **-G***n*) with an initial value	Initialised data segment
.data.*	Large initialised variables compiled with **–fdata–sections**	
.gnu.linkonce.d.*	C++ “link-once” data	
.lit4	Constants (usually floating point) which the assembler decides to store in memory rather than in the instruction stream	Small initialised data segment
.lit8		
.sdata	Variables <=*n* bytes (compiled **-G***n*) with an initial value	
.sdata.*	Small variables compiled with **–fdata–sections**	
.gnu.linkonce.s.*	C++ “link-once” small data	
.sbss	Uninitialised variables <=*n* bytes (compiled **-G***n*)	Small zero-filled segment
.sbss.*	Small uninitialised variables compiled with **–fdata–sections**	
.gnu.linkonce.sb.*	C++ “link-once” small uninitialised data	
.bss	Uninitialised larger variables	Zero-filled segment
.bss.*	Uninitialised variables compiled with **–fdata–sections**.	
.gnu.linkonce.b.*	C++ “link-once” uninitialised data	
.debug*	DWARF debug information	Not in load image
.line		
.stab*	Stabs debug information	
.comment	#ident/.ident strings	
.gptab.*	Information section	
.reginfo	Information section	

Named ELF sections also exist to hold relocation records, symbol tables, etc, but they don’t show up in the final program at all.

ELF is unnecessarily complicated for ready-to-run programs; program loaders and ROM converters would like a simpler format. So the linker can be asked to attach a *program header* to a fully-linked program; the header tells the loader which chunks of the file matter, and where they should go. This is referred to as the “*Execution View*” – the gory details shown in Table 16-1 “Standard ELF section names” are called the “*Linking View*”.

If you need to write code which reads the “Execution View” of an ELF file, perhaps to create your own file loader, then you could look at the SDE *zload* example program, or at the `convert` directory in the SDE tool source tarball.

16.3 ECOFF Object File Format

SDE gives some support to systems using the historical MIPS Computer Systems ECOFF object file format.

- *sde−ld* can incorporate ECOFF object files to produce ELF executables. This works only when producing fully-resolved programs, but does allow you to use old ECOFF libraries. Note that ECOFF libraries start with identical headers to ELF libraries – so you need to tell *sde−ld* explicitly about the type of the file, e.g. put the **−b ecoff−bigmips** or **−b ecoff−littlemips** option in front of the ECOFF library.
- *sde−ld* can also produce an ECOFF executable from a mixture of ELF and ECOFF input files.

Note that neither debugging information nor relocation records (which may be required for a multi-stage link) survive conversion between object formats.

16.4 Using Extra Sections

The compiler and assembler already generate a multitude of different object file *sections* which get linked together into (typically) three large output *segments*: read-only code & data, initialised data, and uninitialised (zero) data – as shown in Table 16-1 "Standard ELF section names" above.

In some applications it may be necessary to define additional object code sections and segments which can be located at disjoint areas within the CPU's address map. We'll take as an example an M4K CPU core. This core has no cache, but a high-speed on-chip *ISRAM* (Instruction SRAM) at a fixed virtual address, say `0x0`. With a CPU like this you would want to locate certain critical functions within the SPRAM region, but you would have to blow them into a PROM at a different address, which would then be copied to the ISRAM at run-time.

16.4.1 Assembler Section Definition

New sections are introduced to the assembler by the following directive:

```
.section name,"flags",@progbits[,align]
```

The section *name* can be any symbol, but by convention begins with a dot. The *flags* are a string of 0 to 4 characters selected from:

Table 16-2 Section attribute flags

Flag	*Meaning*
a	allocate address space
w	contains writable data
x	contains executable instructions
g	contains gp-accessible data

The optional final *align* parameter specifies the required section alignment, as a power-of-two. So, to introduce a code section intended for on-chip SRAM we could use the following:

```
.section .isram, "ax", @progbits, 2
```

After this initial definition has been seen by the assembler, you can then omit all but the section name, e.g. "`.section .isram`".

The assembler remembers the previous section (beware, it's only a one-level stack!), and you can return to it using `.previous` directive.

16.4.2 C/C++ Section Definition

Segment switching in C or C++ is quite different; the compiler already has to keep track of sections and emits section directives as necessary. If you want to steer some particular piece of data or code into a particular named section, then GNU C provides an "`attribute()`" extension mechanism, for example:

```
/* put variable foo into section .xdata */
__attribute__ ((section (".xdata"))) int foo;

/* put function bar into section .isram */
__attribute__ ((section (".isram")))
int bar ()
{
    return 1;
}
```

But often what you want to do is collect a group of functions (or a group of data) into some special fixed area of the CPU memory map, so it may be more convenient to be able to decide which functions to group together at link time instead of compile time.

One way to do this is to give each function its own unique section name, and then generate a linker script which combines only the ones which we want into the hardware-significant segment. The compiler's **–ffunction–sections** option outputs each function into a section whose name is straightforwardly based on the function's name (i.e. `.text.`*`fnname`*) – you can then manipulate the individual function sections at link time, and functions not assigned to a specific output segment will simply be merged into the global `.text` section.

See the **––function–ordering** option in the [Gprof] manual for another way to order individual functions, based on profiling data. See also Section 12.2.2 "Optimizing for Size" for another use of unique function sections, to reduce code size.

If the functions that you move in this way end up out of reach of the normal `jal` instruction (which is restricted to operating in a 256Mbyte "segment" of memory), then you will have to tell the compiler to use indirect `jalr` instructions to call these functions. See Section 16.4.5 "Calling Remote Functions" for details of how to do this.

16.4.3 Linking Extra Sections

When using non-standard sections you'll have to create your own linker script, see Section 16.1 "Linker Scripts". For the ISRAM example discussed above you might expect to add something like the following lines to the default script:

```
.isram 0x0 :
        { *(.isram) }
```

These lines merge all the *.isram* input sections into the *.isram* output section, located at virtual address `0x0`. The resulting executable module could then be converted into ASCII and downloaded by the board's PROM monitor.

However, when creating a rommable program, your program will have to contain code to copy the *.isram* section from ROM to ISRAM itself. In this case your linker script might contain the following:

```
OVERLAY 0x0 : AT (0xbfc3c000)
{
   .isram { *(.isram) }
}
```

The `AT` directive specifies that although the "overlay" is linked to be run at virtual address `0x0`, it will be positioned at address `0xbfc3c000` in the load image (the load address). The load address in this case is the top 4KB of a 256KB boot PROM (base address `0xbfc00000`). Your startup code must then copy the code from this known address into the ISRAM, e.g.

```
extern char __load_start_isram[];
extern char __load_stop_isram[];
/* copy from load address to run address */
isram_write (0x0, __load_start_isram,
               __load_stop_isram - __load_start_isram);
```

If you have a number of C modules which contain code only intended for ISRAM (as described in the previous section), then you can name them explicitly in the script here, e.g.

```
OVERLAY 0x0 : AT (ALIGN(_etext, 16) + (_edata - _fdata))
{
   .isram {
           *(.isram)
           c_isram1.o(.text)
           c_isram2.o(.text)
         }
}
```

This example will include the `.text` section (i.e. code) from files `c_isram1.o` and `c_isram2.o`, and merge them into the output `.isram` section. The `.text` sections from all other object files listed on the linker command line will be handled in the normal way.

Note the more complex `AT` expression in this example. When you use the `sde-conv` program to create a PROM image, it rearranges the sections, and places a copy of the initialised data sections at the next 16 byte boundary after the code (from where the ROM startup code copies it to RAM). This example places the ‘‘overlay’’ code immediately after the initialised data in the ROM.

Please contact us for sample linker scripts, if this short description does not answer your needs.

16.4.4 Linker Garbage Collection

In Section 12.2.2.1 ‘‘Code and data garbage collection’’ we showed how you can use the compiler’s **–ffunction–sections** and **–fdata–sections** options, with the linker’s **–gc–sections** option, to remove unused code and data from your application.

This process of linker ‘‘garbage collection’’ may require some manual intervention if there are sections of your code or data which are not explicitly referenced by your code, but are perhaps required by some external software, such as an operating system loader. In thise case you will have to create a linker script, and mark those sections which must not be eliminated using the ‘‘`KEEP`’’ directive, for ecample ‘‘`KEEP(*(.init))`’’. See the linker manual [Ld] for more details.

Note that **–gc–sections** cannot be used when generating a relocatable output file, i.e. when using the linker’s **–r** flag.

16.4.5 Calling Remote Functions

Although data in additional sections can be accessed without any special precautions, care must be taken when calling functions in them. The MIPS call (`jal`) instruction can’t specify a full 32-bit target (MIPS instructions are only 32 bits long, and there has to be an opcode field to identify this instruction...); instead, it stores 28 bits of the target address; the high 4 bits of the target address are just those of the `jal` instruction. The effect is that you can only call a function in the same 512Mbyte ‘‘page’’ of memory; the linker will complain if you attempt to reach further.

There are ways around this problem:

1) In C you can declare the remote function using the *longcall* or *far* attribute, e.g.:

```
extern int far_away () __attribute__((longcall));
```

or

```
extern int far_away () __attribute__((far));
```

2) In assembler you must explicitly take the address of the function before calling it, e.g.:

```
la      t8,remfunc
jalr    t8
```

3) For C code where changing the source is not possible, you can compile with the **–mlong–calls** option. This forces the compiler to default to performing all function calls using the two-step `la/jalr` sequence. Note that this incurs a speed and space penalty, as ALL function calls will now require at least three instructions instead of one.

To avoid the extra overhead when you know that certains function can be reached with an absolute 28-bit address, you can mark such functions with the *near* attribute, e.f.

```
extern int close_by () __attribute__((near));
```

Manual Downloading

Once the linker has generated an executable object file you may want to download it manually to a PROM programmer, or an evaluation board...

17.1 Evaluation Board Download

Usually you'll download your code using *sde−gdb* as part of a debugging session, as described in the previous chapter. But sometimes you might need to download your program manually. There are usually two steps:

1) While some evaluation boards have an Ethernet interface which allows them to load object files directly at very high speed, most others require that the object file is first converted into some other format (ASCII or encoded binary). The *sde−conv* program performs the task of converting an executable object file into a number of different formats, including: Motorola S-records, LSI Logic PMON fast format, IDT/sim binary S-records, and Stag PROM programmer binary format. See [Conv] for full option details.

   Remember that the example *makefiles* automatically generate downloadable files as their final result. See Section 9.2 "Example Makefiles" for more details.

2) Finally you can can perform the download via a serial or parallel port. It may also be possible to use the download features of your favourite terminal emulator, for which consult your board manual.

Note that when you download to an evaluation board, you will usually want the program and its data to be loaded at the load addresses assigned by the linker, so **DO NOT** use *sde−conv*'s **−p** (prom) option to create your downloadable file: this is what the example *makefiles* will do when building the *ram* and standalone versions of a program, as opposed to the *rommable* version.

The actual process of downloading to an evaluation board is highly dependent on the board and its PROM monitor.

17.2 PROM Programmer Download

The other situation when manual downloading is required is when blowing a PROM. In this case it is usually necessary for the code and data to be relocated from their linker-assigned addresses into offsets from the start of the ROM. The ROM startup code will then relocate the initialised data, and possibly the code too, from ROM to RAM.

The *sde−conv* **−p** (prom) option helps with this. It ensures that ROM resident code and read-only data is placed at its correct offset in the ROM image, and then places the initialised, writable data segment at the next 16-byte boundary following. This supports the behaviour of SDE's default ROM startup code (`romlow.sx`), which copies the initialised data to its final location in RAM before starting your application. See Section 21.4.1 "CPU Reset Handling" for details. *Sde−conv* also contains facilities for splitting an object file into horizontal and/or vertical slices, including interleaving, to accommodate dumb programmers (the machines, not the people!).

The example *makefiles* automatically invoke *sde-conv* with the **−p** option when building *rom* versions of the program.

The physical process of downloading to the PROM programmer is device-dependent. You should refer to your PROM programmer's manual for instructions.

17.3 Other Techniques

Downloading large programs via a serial port is very slow and tedious. There is no reason why a faster technique cannot be used for downloading the program, and you may want to use some other high-speed mechanism on your own board (e.g. a Centronics parallel interface, a PCI bus, USB, or whatever).

To help with this process you may want to examine the sources of *convert* (aka *sde−conv*) programs in the source code tarball.

Intrinsics for MIPS® Architecture

The MIPS architecture includes a number of instructions and registers that can't be accessed directly by C and C++ code. SDE includes a set of *intrinsics* which provide access to these special purpose instructions. They are often implemented in header files, using *gcc* inline *asms* – which means that you can read, modify and reuse them for your own purposes.

This chapter describes only application-level MIPS intrinsics – for intrinsics which access a CPU's "system" facilities see Section 20.6 "System Coprocessor (CP0) Intrinsics".

18.1 Intrinsics for Byte Swapping

Include the header file *<sys/endian.h>* to define the following inline functions. On a MIPS32 Release 2 CPU they will generate a fast two instruction sequence; on other MIPS ISAs they will generate a longer sequence of shifts, ands and ors. They are also smart enough to byte-swap constants at compile time.

`uint32_t htobe32(uint32_t val)`
: Convert the 32-bit value `val` from "host" byte order to big-endian byte order (this will be a no-op on a big-endian CPU).

`uint16_t htobe16(uint16_t val)`
: Convert the 16-bit value `val` to big-endian format.

`uint32_t betoh32(uint32_t val)`
: Convert 32-bit big-endian value `val` to the "host" byte order (this will be a no-op on a big-endian CPU).

`uint16_t betoh16(uint16_t val)`
: Convert 16-bit big-endian value `val` to the "host" byte order.

`uint32_t htole32(uint32_t val)`

`uint16_t htole16(uint16_t val)`

`uint32_t letoh32(uint32_t val)`

`uint16_t letoh16(uint16_t val)`
: As above, but converting to and from little-endian.

18.2 Intrinsics for MIPS32® Architecture

The MIPS32 and MIPS64 instruction set architectures include the count-leading-zeroes and count-leading-ones instructions. SDE provides this C interface, implemented by inline *asm*s on MIPS32 & MIPS64 CPUs, or as a subroutine call on older MIPS architectures. To use these functions include the header file *<mips/mips32.h>*.

`uint32_t mips_clz(uint32_t val)`
: The 32-bit argument `val` is scanned from most significant to least significant bit, and the number of leading zeros is returned. If no bits were set then the value 32 is returned.

`uint32_t mips_clo(uint32_t val)`
: The 32-bit argument `val` is scanned from most significant to least significant bit, and the number of leading ones is returned. If all bits were set then the value 32 is returned.

`uint32_t mips_dclz(uint64_t val)`
: The 64-bit argument `val` is scanned from most significant to least significant bit, and the number of leading zeros is returned. If no bits were set then the value 64 is returned.

`uint32_t mips_dclo(uint64_t val)`
: The 64-bit argument `val` is scanned from most significant to least significant bit, and the number of leading ones is returned. If all bits were set then the value 64 is returned.

18.3 Intrinsics for MIPS32® Release 2 Architecture

The MIPS32 Release 2 ISA introduces a number of new user-level instructions. Some of them will be happily used by the compiler to optimize normal C code, as desribed in Section 12.1.2 "Instruction Set Flags". But some of the byte- and bit-shuffling instructions are not available normal C code, so these intrinsics are made available by including <*mips/mips32.h*>:

`uint32_t _mips32r2_bswapw(uint32_t int val)`
: Byte swap the 32-bit value `val`, a two instructions sequence. It is normally more efficient to use the intrinsics described in Section 18.1 "Intrinsics for Byte Swapping".

`uint32_t _mips32r2_wsbh(uint32_t val)`
: Return the result of the MIPS32 Release 2 `wsbh` instruction given `val`.

`uint32_t _mips32r2_ins(uint32_t tgt, uint32_t val, uint32_t pos, uint32_t sz)`
: Return the result of a 32-bit insert bit field instruction, inserting `sz` bits of `val` into `tgt`, at bit position `pos`. Both `pos` and `sz` must be constants.

`uint32_t _mips32r2_ext(uint32_t x, uint32_t pos, uint32_t sz)`
: Return the result of a 32-bit unsigned extract bit field instruction, returning `sz` bits, from bit position `pos`, of `x`. Both `pos` and `sz` must be constants.

18.4 Intrinsics for MIPS64® Release 2 Architecture

The MIPS64 Release 2 ISA inherits the MIPS32 Release 2 instructions and their intrinsics, but (as one would expect) adds some 64-bit equivalents:

`uint64_t _mips64r2_bswapd(uint64_t val)`
: Byte swap the 64-bit value `val`, a two instructions sequence.

`uint64_t _mips64r2_dsbh(uint64_t val)`
: Return the result of the MIPS64 Release 2 `dsbh` instruction given `val`.

`uint64_t _mips64r2_dshd(uint64_t val)`
: Return the result of the MIPS64 Release 2 `dshd` instruction given `val`.

`uint64_t _mips64r2_dins(uint64_t tgt, uint64_t val, uint32_t pos, uint32_t sz)`
: Return the result of a 64-bit insert bit field instruction, inserting `sz` bits of `val` into `tgt`, at bit position `pos`. Both `pos` and `sz` must be constants.

`uint64_t _mips64r2_dext(uint64_t x, uint64_t pos, uint32_t sz)`
: Return the result of a 64-bit unsigned extract bit field instruction, returning `sz` bits, from bit position `pos`, of `x`. Both `pos` and `sz` must be constants.

18.5 Intrinsics for CorExtend™ Extension

MIPS Technologies' Pro Series™ CPU cores include the CorExtend™ feature, which extends the instruction set by adding a small number of user definable instructions (UDI). The Pro Series cores then provide an on-chip interface which allows a customer building a SoC to add just the logic to implement their chosen instructions; the interface to the CPU pipeline and its general-purpose registers is provided by the core.

The UDI instructions commonly have the standard MIPS "three-operand" format, where they can use two registers as source operands and one as destination[19]. Instructions which don't use all the possible general purpose registers can recycle the register fields for other purposes.

The assembler interface to UDI provides you with choices about how you construct the instruction:

`udi IMM :`
: All 24 user definable bits of the instruction are set by integer `IMM`, including the register and opcode fields.

`udiOP IMM :`
: `OP` is an integer (0 to 15) which defines the UDI opcode, and `IMM` the remaining 20 user-definable bits.

`udiOP rs,IMM :`
: `OP` is the UDI opcode, `rs` the register number (read-only, or read-write), and `IMM` the remaining 15 bits.

`udiOP rs,rt,IMM :`
: `Rs` would conventionally be read-only, but `rt` read-only or read-write. `IMM` is the remaining 10 bits.

`udiOP rs,rt,rd,IMM :`
: `Rs` and `rt` would conventionally be read-only, and `rd` write-only, a conventional MIPS three-operand instruction, with `IMM` defining the remaining 5 bits.

If a register field in a UDI instruction isn't a general purpose register, but a register in the UDI block, or extra opcode bits, then use the `$n` syntax to insert a 5-bit immediate into the field, e.g. **`udi3 $a0,$10,$v0,12`**.

In SDE you get a C interface to the UDI instructions; you'll need to `#include <mips/udi.h>`.

The GNU compiler can optimize code around the `asm()` statements used to build this interface; and that's great. But some UDI instructions may alter internal state or registers in the UDI block which aren't visible to the compiler, making those optimizations incorrect. If your UDI instruction generates no state except for what it writes to the CPU destination register, then you can use the "safe" intrinsics, and the optimizer can work its magic.

In the description below `OP` is the UDI opcode (0 to 15); `A` and `B` are any valid C or C++ scalar integer-valued expression, and `IMM` is a constant to fill the remaining instruction bits. The compiler allocates registers to hold the `A` and `B` source operands, and the result register.

[19] The two source registers are decoded inside the CPU core, and sent to the customer's UDI block, and so they can only be encoded in the standard position. The register number to which to write the result is selected by the UDI block, so in principle can be any CPU register or none, including one of the source registers; but it would be eccentric and unhelpful to specify a separate destination register and not use the standard MIPS format to do it.

```
/* Simple UDI instructiions are assumed to write a result to their
   final CPU register operand, but may may have other side effects
   such as using or modifying internal UDI registers, so they won't be
   optimized by the compiler. */

/* The 'ri' single register intrinsic passes A in the RS field, and
   returns the new RS register. IMM is the remaining 15 bits. */
typeof A mips_udi_ri (OP, A, IMM);

/* The 'rwi' two register intrinsic passes A in the RS field, and
   and returns the new RT register. IMM is the remaining 10 bits. */
typeof A mips_udi_rwi (OP, A, IMM);

/* The 'rri' two register intrinsic passes A in RS, B in RT, and
   and returns the new RT register. IMM is the remaining 10 bits. */
typeof A mips_udi_rri (OP, A, B, IMM);

/* The 'rrwi' three register intrinsic passes A in RS, B in RT,
   and returns the w/o RD register. IMM is the remaining 5 bits. */
typeof A mips_udi_rrwi (OP, A, B, IMM);


/* Optimizable intrinsics for UDI instructions which read only the CPU
   source registers and write to the destination CPU register only,
   and have no other side effects, i.e. they only use and modify the
   supplied CPU registers. */
typeof A mips_udi_ri_safe (OP, A, IMM);
typeof A mips_udi_rwi_safe (OP, A, IMM);
typeof A mips_udi_rri_safe (OP, A, B, IMM);
typeof A mips_udi_rrwi_safe (OP, A, B, IMM);


/* The mips_udi_i() intrinsics use no register inputs, but return the
   value written to the RS register (the input value is assumed
   discarded). */
uint32_t mips_udi_i (OP, IMM);
uint64_t mips_udi_i_64 (OP, IMM);

/* "NoValue" intrinsics for UDI instructions which don't write a
   result to a CPU register, so presumably must have some other side
   effect, such as modifying an internal UDI register. */
void mips_udi_nv (IMM);
void mips_udi_i_nv (OP, IMM);
void mips_udi_ri_nv (OP, A, IMM);
void mips_udi_rri_nv (OP, A, B, IMM);
```

To provide even more flexibility, the following set of intrinsics allow register fields in the UDI instructions to be set to constant 5-bit immediates (0-31), possibly to identify registers inside the UDI block, or as extra opcode bits. The `IS`, `IT` and `ID` arguments below must be constants, which will get inserted into the *rs*, *rt* and *rt* field of the instruction, as appropriate. Arguments `A` and `B` will still be computed and assigned to registers by the compiler.

UDI instructions are allowed to write to any general purpose register, not just those named in the instruction – so the destination register may be implicit in the opcode. To handle this the `GPDEST` argument allows the programmer to explicitly specify the general purpose register number that is written, and this prevents the compiler from allocating that register for other variables across the UDI instruction; if no general purpose CPU register is written, pass a GPDEST of zero.

```
/* These 4 variants of the three register operand format allow
   constant values to be placed in the RS, RT fields, presumably
   because they name internal UDI registers. The RD register is still
   allocated by the compiler. They are implicitly "unsafe" or
   volatile. */
typeof A  mips_udi_riri (OP, A, IT, IMM);
typeof B  mips_udi_irri (OP, IS, B, IMM);
int32_t   mips_udi_iiri_32 (OP, IS, IT, IMM);
int64_t   mips_udi_iiri_64 (OP, IS, IT, IMM);


/* These 5 variants of the three register format allow constant values
   to be placed in the RS, RT and RD fields, presumably because they
   name internal UDI registers. In case the instruction writes to an
   implicit gp register, pass the register number as GPDEST and the
   compiler will be told that it's been clobbered, and its value will
   be returned - if no gp register is written, pass 0. They are all
   implicitly unsafe, or volatile. */
typeof A  mips_udi_rrii (OP, A, B, ID, IMM, GPDEST);
typeof A  mips_udi_riii (OP, A, IT, ID, IMM, GPDEST);
typeof B  mips_udi_irii (OP, IS, B, ID, IMM, GPDEST);
int32_t   mips_udi_iiii_32 (OP, IS, IT, ID, IMM, GPDEST);
int64_t   mips_udi_iiii_64 (OP, IS, IT, ID, IMM, GPDEST);
```

Warning: The compiler assumes that all `asm` inputs are ‘‘word sized’’, i.e. that the inputs have the same precision as the underlying register size, and it may emit instructions to sign- or zero-extend any inputs which are smaller than that (e.g. *char* and *short* operands). To avoid an excessive number of these extension instructions you should try to ensure that you always pass ‘‘word size’’ values to these intrinsics.

Warning 2: The GCC `asm` statement does not allow you to use aggregate values (a *struct*, *union* or array) as inputs or output for an `asm` – you may only pass simple scalar values. If you need to pass aggregate values to or from a UDI instruction, then you must define a *union* to smuggle them through. For example:

```
/* object manipulated by UDI hardware */
typedef struct {
    uint16_t imag;
    uint16_t real;
} complex_t;

/* access mechanism for UDI intrinsics */
typedef union {
    complex_t  c;
    uint32_t   w;
} udicomplex_t;

/* add two complex types using three operand UDI instruction */
extern inline complex_t do_ADDC (const complex_t *a, const complex_t *b)
{
    const udicomplex_t *ua = (udicomplex_t *) a;
    const udicomplex_t *ub = (udicomplex_t *) b;
    udicomplex_t uv;
    uv.w = mips_udi_rrwi_safe (ADDC_OPCODE, ua->w, ub->w, 0);
    return uv.c;
}
```

18.6 Intrinsics for COP2 Extension

Some MIPS Technologies CPU cores allow a SoC builder to design a tightly-coupled coproprocessor which implements the COP2 instructions. These instructions are a part of the MIPS32 and MIPS64 ISAs reserved for use only by coprocessors. For the C interface to these instructions you must `#include <mips/cop2.h>`, which defines the following intrinsics:

`void mips_lwc2 (C2REG, MEM);`
Load the 32-bit word in memory referenced by `MEM` into COP2 data register C2DREG (constant 0-31). The form of MEM is basically a 32-bit value obtained through a pointer, as in:

```
int *a;
mips_lwc2 (3, *a)
```

It's there so you can load a memory value directly into a COP2 register without loading it first into a general-purpose register.

`void mips_swc2 (C2DREG, MEM);`
The opposite – store COP2 data register C2REG to a memory location.

`void mips_ldc2 (C2DREG, MEM);`

`void mips_sdc2 (C2DREG, MEM);`
64-bit load/store respectively. Particularly important if your CPU has only got 32-bit general purpose registers.

`void mips_mtc2 (VAL, C2DREG, SEL);`
Write any 32-bit C expression `VAL` to COP2 register `C2DREG` in register bank `SEL`.

`uint32_t mips_mfc2 (C2DREG, SEL);`
Return the 32-bit COP2 register `C2DREG/SEL`.

`void mips_dmtc2 (VAL, C2DREG, SEL);`

`uint64_t mips_dmfc2 (C2DREG, SEL);`
64-bit versions of the above.

`void mips_ctc2 (VAL, C2CREG);`
Write any 32-bit C expression `VAL` to COP2 control register `C2CREG`.

`uint32_t mips_cfc2 (C2CREG);`
Return the 32-bit COP2 control register `C2CREG`.

`void mips_cop2 (OP);`
Emit arbitrary coprocessor 2 instruction with "undefined" bits set by constant integer `OP`.

`int mips_c2t (CC);`
Returns one if coprocessor 2 condition bit `CC` (0-7) is "true", zero otherwise.

`int mips_c2f (CC);`
Returns one if coprocessor 2 condition bit `CC` is "false", zero otherwise.

18.7 Intrinsics for SmartMIPS® ASE

MIPS Technologies' 4KSc and 4KSd CPU cores implement the SmartMIPS ASE (application specific extension) to the base MIPS32 instruction set. The bit-rotate and indexed load instructions will be used automatically by the compiler when you use the **–msmartmips** compiler option, see Section 12.1.2 "Instruction Set Flags". The other new instructions may be used from C code by using the C intrinsics defined by `#include <mips/smartmips.h>`, as follows:

`int mips_multp (int a, int b)`
: Return the low 32-bit result of the polynomial-basis multiplication of the two 32-bit binary polynomial arguments `a` and `b`.

`int mips_maddp (int acc, int a, int b)`
: Return the low 32-bit result of the polynomial-basis multiplication of arguments `a` and `b`, polynomially added to `acc`. This can be used with `mips_multp` to construct a polynomial multiply-add loop which can be optimized by the compiler. For example:

```
int
maddp_arr (int *arr, int narr, int factor)
{
    int acc, i;
    acc = mips_multp (arr[0], factor);
    for (i = 1; i < narr; i++)
        acc = mips_maddp (acc, arr[i], factor);
    return acc;
}
```

`int mips_maddp2 (int a, int b)`
: Like `mips_maddp`, but assumes that you've already loaded the accumulator (the `LO` register) in some other way that is not visible to the compiler.

`long long mips_multpx (int a, int b)`

`long long mips_maddpx (long long acc, int a, int b)`

`long long mips_maddp2x (int a, int b)`
: Like `mips_multp` etc, but operating on the full 64-bit multiplier result, i.e. the `HI`, `LO` register pair.

`int mips_mfxu (void)`
: Return the extra high order bits (bits 64 and upwards) of the multiply accumulator register (the new SmartMIPS `ACX` register). This is destructive of the accumulator, so use with care.

`int mips_mfhu (void)`
: Return bits 32-63 of the multiply accumulator (the `HI` register). This is destructive.

`int mips_mflhxu (int acc, int &lo)`
: Stores the low 32-bits of the multiply accumulator in `acc` into the lvalue "reference" argument `lo`, and then shifts the multiply accumulator right by 32-bits, returning the shifted accumulator. For example:

```
unsigned int
mpmadd (unsigned int *arr, unsigned int *spill, int narr, int factor)
{
    unsigned int acc = 0;
    int i, j;
    for (i = j = 0; i < narr; i += 4, j++) {
        acc += arr[i+0] * factor;
        acc += arr[i+1] * factor;
        acc += arr[i+2] * factor;
        acc += arr[i+3] * factor;
        acc = mips_mflhxu (acc, spill[j]);
    }
    return acc;
}
```

`long long mips_mflhxux (long long acc, int &lo)`
Like `mips_mflhxu` etc, but operating on the full 64-bit multiplier result, i.e. the `HI`, `LO` register pair.

`void mips_mtlhx (int lo, int hi, int ex)`
Moves the three 32-bit values in arguments `lo`, `hi`, and `ex` to the multiplier result registers (`LO`, `HI` and `ACX`).

`void mips_pperm (int src, int sel)`
Shift the 96-bit (max) extended multiplier result registers 6 bits left, and mix in 6 bits of `src`, permuted by `sel`. See the SmartMIPS **`pperm`** instruction definition for details.

18.8 Intrinsics for Paired-single / MIPS-3D® Architecture

This version of GCC includes support in the compiler for the paired-single SIMD floating point data type and instructions, and the MIPS-3D ASE. Full details of the vector data types and intrinsics can be found in the *Target Builtins* section of the [Gcc] reference manual.

18.9 Intrinsics for MIPS® MT ASE

The new instructions introduced by the MIPS MT ASE may be accessed from C code using the intrinsics defined by `#include <mips/mt.h>`, as follows:

`unsigned int mips_mt_fork (void *addr, unsigned int pv, unsigned int cv)`
Fork to `addr`, returning `pv` to parent and `cv` to child.

`unsigned int mips_mt_yield (unsigned int yq)`
Yield with qualifier `yq`, returning active signals.

`int mips_mt_dmt (void)`
Disable MT, returning old enable state.

`int mips_mt_emt (void)`
Enable MT, returning old enable state.

`int mips_mt_dvpe (void)`
Disable multi-VPE mode, returning old enable state.

`int mips_mt_evpe (void)`
Enable multi-VPE mode, returning old enable state.

Other functions in this header file provide access to the new Coprocessor 0 registers provided by the MT ASE, and to registers within other thread contexts. See Section 20.6 "System Coprocessor (CP0) Intrinsics" for a listing.

18.10 Intrinsics for MIPS® DSP ASE

The MIPS DSP ASE defines a set of new instructions to improve the performance of DSP and "Media" applications.

Many of these new DSP instructions operate on Q15 or Q31 fractional data. Q31 is a 32-bit fixed-point fraction which can represent numbers between -1 and very nearly 1, and Q15 is a similar 16-bit fraction. The DSP ASE's favourite 8-bit quantity is an unsigned fraction representing numbers between `0` and `255/256`.

Vectors of 4 × unsigned bytes or 2 × Q15 fractions fit into a 32-bit register, and the DSP ASE includes instructions which operate on all members of a vector at once. For detailed information about the MIPS DSP ASE (and a proper description of fractional data types), see the MIPS DSP ASE documentation [MD00374].

Addition and subtraction on fractional data are really the same as addition and subtraction with unsigned integer data, but multiplication requires a post-multiply shift to align the resulting values appropriately. The new multiply instructions in the DSP ASE that operate on fractional data provide this shift operation.

We do not (yet) have a compiler which knows about fractions. Q15 is an alias for a signed 16-bit integer (`short`), and Q31 is an alias for a signed 32-bit integer (`int`).

This document describes some new vector data types and built-in "intrinsic" functions available under the GNU C compiler. Each instruction in the DSP ASE has its own intrinsic, so you can write anything in C.

To tell GCC to compile for a CPU with DSP ASE support, pass the compiler the **–mdsp** flag.

Vector data types

Some typedefs:

```
typedef v4q7 __attribute__ ((mode(V4QI)));
typedef v2q15 __attribute__ ((mode(V2HI)));
typedef v4i8 __attribute__ ((mode(V4QI)));
typedef v2i16 __attribute__ ((mode(V2HI)));
```

`v2i16`
: a vector of two 16-bit integers.

`v4i8`
: a vector of four 8-bit integers.

`v4q7`
: a vector of four Q7 fractions.

`v2q15`
: a vector of two Q15 fractions.

You can initialize vectors like this:

```
v4i8 a = {1, 2, 3, 4};
v4i8 b;
b = (v4i8) {5, 6, 7, 8};

v2q15 a = {0x0fcb, 0x3a75};
```

> Caution: when the C compiler lets you see inside vectors and other packed data, you see the components in the order they take up in memory when you store the vector. But instructions in the DSP ASE locate vector subcomponents with reference to register bit-numbers. The relationship between bit-numbers and memory addresses changes with the CPU's endianness; so initializers like this are endianness-dependent.

If you're big-endian, then at the C level you'll see the high-bit-number components first – the DSP ASE refers to these as *left* and uses an `l` (letter "l", that is) in instruction names. If you're little-endian, then at the C level you'll see the lower-bit-numbered components first – what the DSP ASE calls *right* using an "r" in the instruction name.

When little-endian, in fact, the one on the left is on the right: perhaps it's better to use a line break between the elements!

To initialize fractional values it's sometimes convenient to do this:

```
v2q15 b;
b = (v2q15) {0.1234 * 32768.0, 0.4567 * 32768.0};
```

The multiplication by `32768.0` effectively pre-shifts the decimal by 15 bits, which is just what you want for a Q15. To initialize a Q31 variable, you need a 31-bit shift, so multiply by `2147483648.0`.

You can use a union type to access vector components. Again, the relationship between the components named in your union and those seen by the DSP ASE will be endianness-dependent.

```
/* 'v4i8' Example  */
typedef union
{
  v4i8 a;
  char b[4];
} v4i8_union;

v4i8 i;
char j, k, l, m;
v4i8_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];
l = temp.b[2];
m = temp.b[3];

/* Assume we want to assign j, k, l, m to i.  */
temp.b[0] = j;
temp.b[1] = k;
temp.b[2] = l;
temp.b[3] = m;
i = temp.a;


/* 'v2q15' Example  */
typedef union
{
  v2q15 a;
  q15 b[2];
} v2q15_union;

v2q15 i;
q15 j, k;
v2q15_union temp;

/* Assume we want to extract from i.  */
temp.a = i;
j = temp.b[0];
k = temp.b[1];

/* Assume we want to assign j, k to i.  */
temp.b[0] = j;
temp.b[1] = k;
i = temp.a;
```

Scalar data types

```
#include <stdint.h>
typedef int32_t q31;
typedef int32_t i32;
typedef uint32_t ui32;
typedef int64_t a64;
```

`q31`
: is really just an alias for a 32-bit signed integer, but an argument or return value with this type reminds you that the data is being interpreted as a Q31 fraction. Same goes for q15.

`i32, ui32`
: are there for C purists, since there's no guarantee that a simple `int` is 32 bits.

`a64`
: is an alias for `long long` (which for MIPS GCC is a 64-bit signed integer). We use it to remind you that the underlying instruction is using one of the four 64-bit accumulators defined by the DSP ASE (`$ac0, $ac1, $ac2, $ac3`). If you're already familiar with the MIPS architecture, note that `$ac0` comprises the bits of the `hi/lo` registers used in regular MIPS32 multiply/divide instructions.

Note that some parameters of builtin function have the following types.

`imm0_7:`
: the parameter must be a constant in the range 0 to 7.

`imm0_15:`
: the parameter must be a constant in the range 0 to 15.

`imm0_31:`
: the parameter must be a constant in the range 0 to 31.

`imm0_63:`
: the parameter must be a constant in the range 0 to 63.

`imm0_255:`
: the parameter must be a constant in the range 0 to 255.

`imm0_1023:`
: the parameter must be a constant in the range 0 to 1023.

`imm1_32:`
: the parameter must be a constant in the range 1 to 32.

`imm_n32_31:`
: the parameter must be a constant in the range -32 to 31.

Compiler builtin functions

The DSP ASE instruction names are full of "." (period) characters, not legal as part of C names. To make C names each period is replaced by "_" (underscore), and the assembler name prefixed with "`__builtin_mips_`".

So the instruction called `addq.ph` becomes `__builtin_mips_addq_ph`. Note that where there are two variants of an underlying DSP instruction which accept an immediate or variable/register operand, the compiler will automatically pick the correct instruction depending on the type and size of the operand.

The instructions are listed in alphabetical order. Spaces have been introduced to separate unlike instructions, but there's no other hint as to what they do.

```
v2q15   __builtin_mips_absq_s_ph (v2q15);
q31     __builtin_mips_absq_s_w (q31);
```

```
v2q15   __builtin_mips_addq_ph (v2q15, v2q15);
v2q15   __builtin_mips_addq_s_ph (v2q15, v2q15);
q31     __builtin_mips_addq_s_w (q31, q31);

i32     __builtin_mips_addsc (i32, i32);
i32     __builtin_mips_addwc (i32, i32);

v4i8    __builtin_mips_addu_qb (v4i8, v4i8);
v4i8    __builtin_mips_addu_s_qb (v4i8, v4i8);

i32     __builtin_mips_bitrev (i32);

i32     __builtin_mips_bposge32 ();

void    __builtin_mips_cmp_eq_ph (v2q15, v2q15);
void    __builtin_mips_cmp_le_ph (v2q15, v2q15);
void    __builtin_mips_cmp_lt_ph (v2q15, v2q15);

i32     __builtin_mips_cmpgu_eq_qb (v4i8, v4i8);
i32     __builtin_mips_cmpgu_le_qb (v4i8, v4i8);
i32     __builtin_mips_cmpgu_lt_qb (v4i8, v4i8);

void    __builtin_mips_cmpu_eq_qb (v4i8, v4i8);
void    __builtin_mips_cmpu_le_qb (v4i8, v4i8);
void    __builtin_mips_cmpu_lt_qb (v4i8, v4i8);

a64     __builtin_mips_dpaq_s_w_ph (a64, v2q15, v2q15);
a64     __builtin_mips_dpaq_sa_l_w (a64, q31, q31);

a64     __builtin_mips_dpau_h_qbl (a64, v4i8, v4i8);
a64     __builtin_mips_dpau_h_qbr (a64, v4i8, v4i8);

a64     __builtin_mips_dpsq_s_w_ph (a64, v2q15, v2q15);
a64     __builtin_mips_dpsq_sa_l_w (a64, q31, q31);

a64     __builtin_mips_dpsu_h_qbl (a64, v4i8, v4i8);
a64     __builtin_mips_dpsu_h_qbr (a64, v4i8, v4i8);

i32     __builtin_mips_extp (a64, i32);
i32     __builtin_mips_extpdp (a64, i32);

i32     __builtin_mips_extr_r_w (a64, i32);
i32     __builtin_mips_extr_rs_w (a64, i32);
i32     __builtin_mips_extr_s_h (a64, i32);
i32     __builtin_mips_extr_w (a64, i32);

i32     __builtin_mips_insv (i32, i32);

i32     __builtin_mips_lbux (void *, i32);
i32     __builtin_mips_lhx (void *, i32);
i32     __builtin_mips_lwx (void *, i32);

a64     __builtin_mips_maq_s_w_phl (a64, v2q15, v2q15);
a64     __builtin_mips_maq_s_w_phr (a64, v2q15, v2q15);
a64     __builtin_mips_maq_sa_w_phl (a64, v2q15, v2q15);
a64     __builtin_mips_maq_sa_w_phr (a64, v2q15, v2q15);
```

```
i32     __builtin_mips_modsub (i32, i32);

a64     __builtin_mips_mthlip (a64, i32);

q31     __builtin_mips_muleq_s_w_phl (v2q15, v2q15);
q31     __builtin_mips_muleq_s_w_phr (v2q15, v2q15);

v2q15   __builtin_mips_muleu_s_ph_qbl (v4i8, v2q15);
v2q15   __builtin_mips_muleu_s_ph_qbr (v4i8, v2q15);

v2q15   __builtin_mips_mulq_rs_ph (v2q15, v2q15);

a64     __builtin_mips_mulsaq_s_w_ph (a64, v2q15, v2q15);

v2q15   __builtin_mips_packrl_ph (v2q15, v2q15);

v2q15   __builtin_mips_pick_ph (v2q15, v2q15);
v4i8    __builtin_mips_pick_qb (v4i8, v4i8);

q31     __builtin_mips_preceq_w_phl (v2q15);
q31     __builtin_mips_preceq_w_phr (v2q15);

v2q15   __builtin_mips_precequ_ph_qbl (v4i8);
v2q15   __builtin_mips_precequ_ph_qbla (v4i8);
v2q15   __builtin_mips_precequ_ph_qbr (v4i8);
v2q15   __builtin_mips_precequ_ph_qbra (v4i8);

v2q15   __builtin_mips_preceu_ph_qbl (v4i8);
v2q15   __builtin_mips_preceu_ph_qbla (v4i8);
v2q15   __builtin_mips_preceu_ph_qbr (v4i8);
v2q15   __builtin_mips_preceu_ph_qbra (v4i8);

v2q15   __builtin_mips_precrq_ph_w (q31, q31);
v4i8    __builtin_mips_precrq_qb_ph (v2q15, v2q15);
v2q15   __builtin_mips_precrq_rs_ph_w (q31, q31);

v4i8    __builtin_mips_precrqu_s_qb_ph (v2q15, v2q15);

i32     __builtin_mips_raddu_w_qb (v4i8);

i32     __builtin_mips_rddsp (imm0_63);

v2q15   __builtin_mips_repl_ph (i32);
v4i8    __builtin_mips_repl_qb (i32);

a64     __builtin_mips_shilo (a64, i32);

v2q15   __builtin_mips_shll_ph (v2q15, i32);
v4i8    __builtin_mips_shll_qb (v4i8, i32);
v2q15   __builtin_mips_shll_s_ph (v2q15, i32);
q31     __builtin_mips_shll_s_w (q31, i32);

v2q15   __builtin_mips_shra_ph (v2q15, i32);
v2q15   __builtin_mips_shra_r_ph (v2q15, i32);
q31     __builtin_mips_shra_r_w (q31, i32);
```

```
v4i8    __builtin_mips_shrl_qb (v4i8, i32);

v2q15   __builtin_mips_subq_ph (v2q15, v2q15);
v2q15   __builtin_mips_subq_s_ph (v2q15, v2q15);
q31     __builtin_mips_subq_s_w (q31, q31);

v4i8    __builtin_mips_subu_qb (v4i8, v4i8);
v4i8    __builtin_mips_subu_s_qb (v4i8, v4i8);

void    __builtin_mips_wrdsp (i32, imm0_63);
```

Compiler builtins for second revision

The second revision of the DSP ASE introduces some new instructions for which there are equivalent new builtin functions in the compiler.

```
v4q7 __builtin_mips_absq_s_qb (v4q7);

v2q15 __builtin_mips_addqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_addqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_addqh_w (q31, q31);
q31 __builtin_mips_addqh_r_w (q31, q31);

v2i16 __builtin_mips_addu_ph (v2i16, v2i16);
v2i16 __builtin_mips_addu_s_ph (v2i16, v2i16);

v4i8 __builtin_mips_adduh_qb (v4i8, v4i8);
v4i8 __builtin_mips_adduh_r_qb (v4i8, v4i8);

i32 __builtin_mips_append (i32, i32, imm0_31);
i32 __builtin_mips_balign (i32, i32, imm0_3);

i32 __builtin_mips_cmpgdu_eq_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_lt_qb (v4i8, v4i8);
i32 __builtin_mips_cmpgdu_le_qb (v4i8, v4i8);

a64 __builtin_mips_dpa_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dps_w_ph (a64, v2i16, v2i16);

a64 __builtin_mips_dpaqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpaqx_sa_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpax_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsx_w_ph (a64, v2i16, v2i16);
a64 __builtin_mips_dpsqx_s_w_ph (a64, v2q15, v2q15);
a64 __builtin_mips_dpsqx_sa_w_ph (a64, v2q15, v2q15);

a64 __builtin_mips_madd (a64, i32, i32);
a64 __builtin_mips_maddu (a64, ui32, ui32);
a64 __builtin_mips_msub (a64, i32, i32);
a64 __builtin_mips_msubu (a64, ui32, ui32);

v2i16 __builtin_mips_mul_ph (v2i16, v2i16);
v2i16 __builtin_mips_mul_s_ph (v2i16, v2i16);

q31 __builtin_mips_mulq_rs_w (q31, q31);
v2q15 __builtin_mips_mulq_s_ph (v2q15, v2q15);
q31 __builtin_mips_mulq_s_w (q31, q31);
a64 __builtin_mips_mulsa_w_ph (a64, v2i16, v2i16);
```

```
a64 __builtin_mips_mult (i32, i32);
a64 __builtin_mips_multu (ui32, ui32);


v4i8 __builtin_mips_precr_qb_ph (v2i16, v2i16);
v2i16 __builtin_mips_precr_sra_ph_w (i32, i32, imm0_31);
v2i16 __builtin_mips_precr_sra_r_ph_w (i32, i32, imm0_31);


i32 __builtin_mips_prepend (i32, i32, imm0_31);


v4i8 __builtin_mips_shra_qb (v4i8, i32);
v4i8 __builtin_mips_shra_r_qb (v4i8, i32);
v2i16 __builtin_mips_shrl_ph (v2i16, i32);


v2q15 __builtin_mips_subqh_ph (v2q15, v2q15);
v2q15 __builtin_mips_subqh_r_ph (v2q15, v2q15);
q31 __builtin_mips_subqh_w (q31, q31);
q31 __builtin_mips_subqh_r_w (q31, q31);


v2i16 __builtin_mips_subu_ph (v2i16, v2i16);
v2i16 __builtin_mips_subu_s_ph (v2i16, v2i16);


v4i8 __builtin_mips_subuh_qb (v4i8, v4i8);
v4i8 __builtin_mips_subuh_r_qb (v4i8, v4i8);
```

18.11 Intrinsics for Atomic R-M-W

SDE includes a set of atomic read-modify-write operations which provide fast, protected access to shared memory locations (but not device registers) in the face of interrupts. In the case of processors which support the `ll` and `sc` instructions, and have the appropriate external hardware, they will also be multi-processor safe. These facilities can be used to implement semaphores, mutexes, counters, etc.

To use these functions include the header file *<mips/atomic.h>*. The functions are as follows:

`uint32_t mips_atomic_bis(uint32_t *wp, uint32_t bits)`
: The atomic bit "test-and-set" operation: sets those bits in `*wp` selected by non-zero bits in `bits` (e.g. `*wp |= set`), and returns the old value of `*wp`.

`uint32_t mips_atomic_bic(uint32_t *wp, uint32_t bits)`
: The atomic bit "test-and-clear" operation: clears those bits in `*wp` selected by non-zero bits in `bits` (e.g. `*wp &= ~clr`), and returns the old value of `*wp`.

`uint32_t mips_atomic_bcs(uint32_t *wp, uint32_t clr, uint32_t set)`
: A combined atomic bit "test-clear-and-set" operation: clears those bits in `*wp` selected by non-zero bits in `clr` and sets those selected by `set` (e.g. `*wp = (*wp & ~clr) | set`). Returns the old value of `*wp`.

`uint32_t mips_atomic_swap(uint32_t *wp, uint32_t new)`
: The atomic "test-and-swap", sets `*wp` to `new`, and returns the old value of `*wp`.

`uint32_t mips_atomic_inc(uint32_t *wp)`
: Atomically increments `*wp`, returning its old value.

`uint32_t mips_atomic_dec(uint32_t *wp)`
: Atomically decrements `*wp`, returning its old value.

`uint32_t mips_atomic_add(uint32_t *wp, uint32_t val)`
: Atomically adds `val` to `*wp`, returning its old value.

`uint32_t mips_atomic_cas(uint32_t *wp, uint32_t new, uint32_t cmp)`
: Atomic "compare-and-swap": sets `*wp` to `new`, but only if it originally equals `cmp`. It returns the original value of `*wp`, whether or not updated.

Note that when the CPU does not include the `ll` and `sc` instructions, the operation is simulated, and will only be atomic if all interrupts are handled by the standard SDE exception handler, where there is special fixup code.

18.12 Intrinsics for Data Prefetch

Some MIPS-Based CPUs support the `pref` instruction, which allows a programmer to optimize array processing loops (as used in many DSP algorithms) by explicitly prefetching the next block of data into the data cache before it is needed, to minimise the cache-miss latency of the following loads and stores. If it is done early enough the data will already be in the cache by the time it is needed.

SDE includes a set of prefetch intrinsics to access these instructions. On CPUs which don't support the `pref` instruction these will be no-ops. To use the intrinsics include the header file *<mips/cpu.h>*.

`void mips_prefetch (void *addr, int rw, int locality)`

The value of `addr` is the address of the memory to prefetch. There are two further arguments: `rw` and `locality`. The value of `rw` is a compile-time constant one or zero; one means that the prefetch is preparing for a write to the memory address and zero means that the prefetch is preparing for a read. The value `locality` must be a compile-time constant integer between zero and three. A value of zero means that the data has no temporal locality, so it need not be left in the cache after the access. A value of three means that the data has a high degree of temporal locality and should be left in all levels of cache possible. Values of one and two mean, respectively, a low or moderate degree of temporal locality. For example:

```
j = mips_dcache_linesize / sizeof (a[0]);
for (i = 0; i < n; i++)
  {
    a[i] = a[i] + b[i];
    mips_prefetch (&a[i+j], 1, 1);
    mips_prefetch (&b[i+j], 0, 1);
    /* ... */
  }
```

Data prefetch does not generate faults if `addr` is invalid, but the address expression itself must be valid. For example, a prefetch of `p->next` will not fault if `p->next` is not a valid address, but evaluation will fault if `p` is not a valid address.

Note that the `mips_prefetch` arguments match the `__builtin_prefetch` intrinsic in GCC 3.x, for which it is an alias.

`void mips_nudge (void *addr)`

The MIPS-specific "nudge"(push to memory) operation. The addressed cache line is written back to memory and invalidated.

`void mips_prepare_for_store (void *addr)`

The MIPS-specific "prepare for store" operation. If the addressed line is not already in the cache, then a line is allocated for it without reading memory (possibly flushing another line from the cache), and the line is cleared to zero. Warning: since this may zero the whole cache line, make sure that you only operate on cache line sized chunks, with cache line alignment.

SDE Run-time I/O System

The SDE run-time system is a library that is built from our Embedded System Kit under the control of a board-specific configuration file. The structure of the source code (for supported SDE customers) is described in Chapter 21 "Embedded System Kit Source ", but this chapter discusses the programming interfaces offered by the library.

The run-time system has two quite distinct parts: a high-level POSIX-like i/o system and environment; plus a collection of low-level CPU management and control primitives. We discuss the POSIX like system in this chapter, and the low-level CPU management in Chapter 20 "CPU Management".

19.1 POSIX API Environment

The C library, described in Section 11.1 "ISO / ANSI C Library", requires a set of low-level, UNIX-like file i/o primitives. The run-time system provides this i/o system, and a *signal* handling mechanism, both of which conform to the POSIX.1 definition. What are the benefits of this?

1) It is a well-documented, and well-known interface, see [POSIX88].
2) It shields the programmer from differences between various PROM monitor or simulator i/o systems. A program can be recompiled unchanged to run on any eval board or simulator supported by SDE.
3) A program will behave identically whether it is running in RAM, under the control of a board's monitor, or standalone in ROM.
4) It makes it very easy to port simple, self-contained programs from UNIX, Linux or other POSIX-compliant systems.

Although we refer you to [POSIX88] for documentation, the remainder of this section describes some of the details specific to this implementation. If your host system supports the POSIX interface (which is true for modern UNIX hosts, and the "Cygwin" environment on Windows) and you have the host's online "manual pages" available, then you'll find that those pages describe most of the functions listed here, and those in the SDE C library.

19.1.1 Remote File I/O

The run-time system implements a read-only POSIX file-system root, which contains a number of named special directories and devices which you can access via the standard POSIX file i/o primitives (e.g. `open`, `close`, `read`, `write`, etc). Note that you cannot cannot create or delete directories and files in this file system, other than as documented below.

19.1.1.1 Host File Access

If your program is running on the GNU simulator, or you're using an MDI connection to your target via *gdb* (e.g. the MIPSsim simulator), then you have access to files on your host computer:

`/host/`*`path`*
: refers to absolute pathname *`path`* on the host computer, e.g. "`/host/etc/passwd`" refers to file `/etc/passwd` on the host.

`/cdir/`*`path`*
: refers to file *`path`* on the host computer, relative to the the debugger or simulator's current directory, e.g. e.g. "`/cdir/Makefile`" will refers to file `Makefile` in your host's current directory.

`/tmp/`*`path`*
: refers to file *`path`* relative to the host's `/tmp` directory.

Furthermore the run-time startup code performs an initial `chdir()` to the `/cdir` directory, so a simple file name without an initial '/' will refer – as you would hope – to a file in the debugger or simulator's current directory – this is handy for benchmark programs which expect to be able read and write their data files in the current directory.

19.1.2 Terminal I/O (/dev/tty)

The pseudo file-system also contains at least the following special device files (some boards may support more):

`/dev/tty0`:
: serial i/o port #0 – the first serial port.

`/dev/tty1`:
: serial i/o port #1 – the second serial port, if present.

`/dev/console`:
: the board's console, usually an alias for `/dev/tty0`.

`/dev/tty`:
: the "controlling terminal", also usually an alias for `/dev/tty0`.

All of these devices support a set of `ioctl` operations which implement the POSIX *termios* interface. These control: input line-editing, output processing, XON/XOFF flow control, baud rate control, "asynchronous" i/o notification, blocking/non-blocking reads, etc. When running under a PROM monitor some of the hardware control `ioctl` operations may have no effect, if they are not accessible via the PROM monitor's API – when running standalone or rommable code they will all be supported, because an SDE serial port driver will have full control of your UART.

Note that the "interrupt" character (default `Ctrl-C`) will raise a POSIX SIGINT signal, but the "quit" character (default `Ctrl-\`) calls the `abort()` function, which will drop you into the debugger.

If you use the non-POSIX `O_ASYNC` flag when you open the tty device (or you use the `fcntl(FASYNC)` function on an open file descriptor), then the SIGIO signal will be raised when an input record is available (although note the comments on polling above).

19.1.3 Linux AP/RP Communication (/dev/lx#)

Programs which are built for targets which use the `mtspmon` "monitor" (currently `MALTA32LSP`, and `MALTA32BSP`), have access to eight character devices named `/dev/lx0-7`. These provide a basic byte stream interface between the SDE "standalone" code running on the Signal Processor side, and the Linux device driver running on the Application Processor side.

The `rtlx` example program demonstrates the use of these devices, see Section 9.1.13 "Linux AP/RP Communication".

19.1.4 Flash Memory Device (/dev/flash)

If your board kit includes support for Flash memory, see Chapter 22 "Retargetting the Toolkit", then there will be special device files with names in the following format:

`/dev/flash`*`N`*`:`
: Where *`N`* is the device number, starting from 0. This file provides access to the whole of the Flash memory device.

`/dev/flash`*`NP`*`:`
: Where *`N`* is the device number, and *`P`* is the *partition* type. Each flash may be divided into a number of sub-partitions, as follows:

Table 19-1 Flash memory partition types

Type	Description
b	Bootstrap (e.g. PROM monitor)
t	Test region (e.g. power-on test scratch area)
e	Non-volatile environment region
f	Data region #1 (e.g. flash file system)
g	Data region #2
h	Data region #3
i	Data region #4

To see whether your board kit supports and has detected Flash memory build and run Section 9.1.3 "Command Line Monitor (minimon)" and use the command "`ls /dev`". You can also display the contents of the Flash using "`dump /dev/flash0`" or similar.

To include the `/dev/flash` interface in your build, you must define `FEATURES=flashdev` or `FEATURES=all` in your application Makefile, see Section 9.2 "Example Makefiles" for details. For a complete example of how to use the interface, see the example program Section 9.1.10 "Flash Memory Test".

Each device can be opened, read and written using the standard POSIX file i/o functions (e.g. `open`, `read`, `write`, `lseek`, etc), and therefore also the buffered *stdio* library functions (`fopen`, `fread`, `fwrite`, `fseek`, etc). This means that you can develop an object file loader, for example, and debug it on a simulator reading from a host file, and then port the code to your target system where it can load from Flash memory. Or the Flash memory might be used as a simple "file" in which to retain configuration data or store log output, and which can be read or written using the stdio library functions like `fscanf` or `fprintf`. A full Flash file system may be provided in future versions of SDE.

Note that although you can write to a Flash device one byte at a time, this will be very slow unless you are writing to an erased region (contains all ones).

The Flash device driver implements the following *ioctls*, as defined in the header file <*sys/flashio.h*>:

FLASHIOINFO

Returns the name (manufacturer and part number) of the Flash device, and its geometry, in the following structure:

```
struct flashinfo {
    char                name[32];       /* dev name */
    unsigned long       base;           /* dev base (phys address) */
    unsigned int        size;           /* dev size */
    unsigned long       mapbase;        /* memory mapped base (phys addr) */
    unsigned char       unit;           /* unit byte size (1,2,4,8 or 16) */
    unsigned int        maxssize;       /* maximum sector size */
    unsigned int        soffs;          /* base offset of specified sector */
    unsigned int        ssize;          /* size of specified sector */
    int                 sprot;          /* specified sector is protected */
}
```

A pointer to this structure is passed as the *ioctl* parameter. If the *soffs* field is set to an offset within the device, then the returned structure will included the base offset of that sector, its size, and its protection status in the *soffs*, *ssize* and *sprot* fields respectively.

Note that when multiple Flash memory devices are organised in parallel banks, then all of the size fields will be multiplied accordingly. For example, if four byte-wide 1 MByte devices are connected in parallel to a 32-bit data bus, then the unit size will be 4 bytes; the sector sizes will be multiplied by 4, and the total device size will be 4 MBytes. If two banks are interleaved then the sizes will be doubled again.

FLASHIOGPART

Returns the type, offset and size of this partition within the whole device, in the following structure:

```
struct flashpart {
    int                 type;           /* partition type */
    unsigned int        offs;           /* base of partition */
    unsigned int        size;           /* size of partition */
}
```

The *type* field is one of the following values:

FLASHPART_RAW
: The whole device.

FLASHPART_BOOT
: The boot partition (e.g. PROM monitor code).

FLASHPART_POST
: Power-on self test (scratch) region.

FLASHPART_ENV
: Non-volatile environment.

FLASHPART_FFS
: Flash file system partition, free for data storage.

FLASHPART_UNDEF
: Undefined type.

FLASHIOGFLGS
: Returns the current device mode which controls how the device is read and programmed. The ioctl parameter should be a pointer to an *int*. The value contains the bitwise OR of the following bits:

  FLASHFLGS_REBOOT
  : Reboot after next write.

  FLASHFLGS_NOCOPY
  : Don't copy programming code to RAM (normally it must be copied if your application is itself executing out of the Flash device).

  FLASHFLGS_MERGE
  : Merge partial sector writes with existing sector data. If this flag is not set then a partial sector write will return an error if you write to a portion of unerased flash.

  FLASHFLGS_CODE
  : Some Flash memories must be programmed differently if they contain executable code, rather than being treated as a simple ''byte stream''.

  FLASHFLGS_STREAM
  : The default mode treats the Flash as a simple sequential byte stream.

  The default value is: FLASHFLGS_MERGE | FLASHFLGS_STREAM.

FLASHIOSFLGS
: Sets the current device mode which controls how the device is read and programmed. The ioctl parameter should be a pointer to an *int* containing the bitwise OR of the flag bits described above.

FLASHIOERASEDEV
: Causes the whole Flash device to be erased. The *ioctl* parameter is ignored. Take care not to use this if your code is running in Flash!

FLASHIOERASESECT
: Erases one Flash device sector. The *ioctl* parameter should be a pointer to an *unsigned int* holding an offset within the sector to be erased.

FLASHIOGPARTS
: The *ioctl* parameter should be a pointer to an array of FLASHNPART *flashparts* structures, as described in FLASHIOGPART above. It will return the complete partition table for this device.

FLASHIOFLUSH
: Forces any pending partial sector writes to written to Flash. This will happen automatically when the device is closed. The *ioctl* parameter is ignored.

19.1.5 Alpha Display (/dev/panel)

If your board kit includes support for an on-board or ‘‘front-panel’’ LED display, then there will be a special device file with the name ‘‘`/dev/panel`’’.

This device can be opened and written using the POSIX file i/o functions (e.g. `open` and `write`), and therefore also the buffered *stdio* library functions (`fopen`, `fprintf`, etc). Each write to the device will by default be automatically preceded by an implicit *seek* to a fixed offset (default zero), and will thus overwrite the last message.

For an example of the use of the `/dev/panel` interface, see Section 9.1.12 ‘‘Decompressing Boot Loader’’.

The panel device driver also implements the following *ioctls*, as defined in the header file <*sys/panelio.h*>:

PANELIOINFO
: Returns information about the display in the following structure:

```
struct panelinfo {
    unsigned char       type;           /* display type */
    unsigned char       flags;          /* display facilities */
    unsigned char       rows;           /* number of rows or lines */
    unsigned char       cols;           /* number of columns per line */
}
```

A pointer to this structure is passed as the *ioctl* parameter. The *type* field will be one of:

PANELTYPE_ALPHA
: Alphanumeric display

PANELTYPE_HEX
: Hexadecimal display

PANELTYPE_LED
: Individual LEDs

The *flags* field describes the capabilities of the display, as the bitwise OR of the following flags:

PANELFLGS_BRIGHTNESS
: The display has variable brightness.

PANELFLGS_CONTRAST
: The display has variable contrast.

PANELFLGS_BLINK
: The whole display can blink on and off.

PANELFLGS_FLASH
: Individual characters or digits can blink.

PANELFLGS_SCROLL
: The display can be scrolled if the message is longer than the display (not currently supported).

PANELFLGS_PROGRESS
: The display has a bar graph or something similar, which can display the progress of a long operation.

PANELIOGMODE
: Returns the current display mode in the following structure:

```
struct panelmode {
    unsigned char       options;        /* display options */
    unsigned char       brighton;       /* brightness (0 to 100%) */
    unsigned char       brightoff;      /* brightness (0 to 100%) */
    unsigned char       contrast;       /* contrast (0 to 100%) */
    unsigned long       blinkon;        /* on period in ns */
    unsigned long       blinkoff;       /* off period in ns */
    unsigned long       scrollrate;     /* scroll rate in ns */
    int                 scrollchars;    /* scroll amount */
}
```

A pointer to this structure is passed as the *ioctl* parameter. The *options* field is the bitwise OR of the following bits:

PANELOPT_PAD
: Pad short messages to the end of the display line with blanks.

PANELOPT_CENTRE
: Centre short messages within each display line.

PANELOPT_WRAP
: Wrap messages longer than one line onto the next line (if available), the default is to truncate the message at the end of the line.

PANELOPT_IGNLF
: Line-feed ('\n') characters will not be treated specially; the default is to cause the following characters to start on the next display line (if available).

PANELOPT_IGNNUL
: NUL characters will not be treated specially, the default is to treat them as the end of the message.

PANELOPT_ROTATE
: Messages longer than one line will continually scroll/rotate. This is not yet supported.

PANELOPT_FADE
: The display brightness will fade up and down, rather than simply flashing/blinking.

PANELOPT_FLASH
: Characters in following writes will be flashed/faded.

PANELIOSMODE
: Sets the current display mode. A pointer to the `panelmode` structure described above is passed as the *ioctl* parameter.

PANELIOCLEAR
: Clear the display. The *ioctl* parameter is ignored.

PANELIOPROGRESS
: Update the panels' bar graph or similar to reflect progress through some long operation. The *ioctl* parameter is a pointer to an *int* with a value between 0 (min) and 100 (max).

PANELIOSCOORD
: Sets the coordinate of the next output message, instead of the default <0, 0>, from the following structure:

```
struct panelcoord {
    unsigned short      row;
    unsigned short      col;
}
```

A pointer to this structure is passed as the *ioctl* parameter. The value is sticky and will be used again on all following writes to the device.

19.1.6 Signal Handling

The run-time system includes an implementation of `sigaction()` and associated signal handling functions defined by [POSIX88], including `sigpending()`, `sigprocmask()`, `sigsuspend()` and `raise()`. Also included is the non-POSIX, but time-honoured UNIX `signal()` function. For an example of how these can be used, see example #3, as described in Section 9.1.3 "Command Line Monitor (minimon)".

For direct access to the lower-level CPU exceptions and interrupts see Section 20.2.1 "C-level Exceptions".

The following is a list of all the signals we use, with names as in the include file <*signal.h*>:

Table 19-2 POSIX signal list

NAME	Default Action	Description
SIGINT	terminate program	interrupt program (^C from terminal)
SIGILL	terminate program	illegal instruction
SIGTRAP	terminate program	debug (breakpoint) trap
SIGABRT	terminate program	abort() call
SIGFPE	terminate program	floating point exception / integer overflow
SIGKILL	terminate program	kill program
SIGBUS	terminate program	bus error or alignment error
SIGSEGV	terminate program	segmentation violation (invalid address)
SIGSYS	terminate program	system call trap
SIGALRM	terminate program	real-time timer expired
SIGIO	ignore signal	I/O is possible on a terminal
SIGVTALRM	terminate program	virtual time alarm (see setitimer() below)
SIGPROF	terminate program	profiling timer alarm (see setitimer() below)
SIGUSR1	terminate program	user defined signal 1
SIGUSR2	terminate program	user defined signal 2

19.1.7 Elapsed Time Measurement

If you need to read the current time, for performance measurement or logging, then see the standard ISO / ANSI `clock()` function, described in [Kern88], which returns the elapsed time in units of 1 microsecond; there is also the `time()` function which returns the current "wall clock" time, in units of 1 second. The <*time.h*> include file defines the following functions like this:

```
clock_t clock (void);
time_t  time (time_t *);
```

Unlike a "real" POSIX operating system, the `clock()` function measures elapsed *real* time, not *cpu* time; in other words it **does** include time spent waiting for console input/output. When measuring performance, be careful to put calls to `clock()` around computational code only.

Alternatively you may prefer to use the POSIX `gettimeofday()` function, which returns the current "wall clock" time in both units and fractions of a second. The <*sys/time.h*> include file defines the following:

```
struct timeval {
        long    tv_sec;                 /* seconds */
        long    tv_usec;                /* and microseconds */
};

struct timezone {
        int     tz_minuteswest;         /* ... of Greenwich */
        int     tz_dsttime;             /* type of DST correction */
};

int     gettimeofday (struct timeval *tvp, struct timezone *tzp);
```

You can pass a null timezone pointer, if you are not interested in that information.

19.1.8 Interval Timing

At the coarsest level, the `alarm(int secs)` function sets an interval timer which expires in `secs` seconds. A SIGALRM signal will be delivered when it expires.

More accurate timing facilities are modelled on those originally provided by POSIX. The <*sys/time.h*> include file defines the following:

```
#define ITIMER_REAL     0
#define ITIMER_VIRTUAL  1
#define ITIMER_PROF     2
#define ITIMER_USER     3

int
getitimer(int which, struct itimerval *value)

int
setitimer(int which, struct itimerval *value, struct itimerval *ovalue)
```

The system provides four separate interval timers. The `getitimer()` call returns the current value for the timer specified by *which* in the structure at *value*. The `setitimer()` call sets a timer to the specified *value* (returning the previous value of the timer if *ovalue* is non-nil).

A timer value is defined by the *itimerval* structure:

```
struct itimerval {
        struct  timeval it_interval;    /* timer interval */
        struct  timeval it_value;       /* current value */
        void    (*it_func)(struct timeval *, struct xcptcontext *);
};
```

If *it_value* is non-zero, it indicates the time to the next timer expiration. If *it_interval* is non-zero, it specifies a value to be used in reloading *it_value* when the timer expires. Setting *it_value* to 0 disables a timer. Setting *it_interval* to 0 causes a timer to be disabled after its next expiration (assuming *it_value* is non-zero).

Note that interval timer values are rounded up to a multiple of 1 millisecond, and that timers are decremented in real time, i.e. no account is taken of whether a program is waiting for i/o or executing useful code.

A SIGALRM signal is delivered when the `ITIMER_REAL` timer expires.

A SIGVTALRM signal is delivered when the `ITIMER_VIRTUAL` timer expires.

A SIGUSR1 signal is delivered when the `ITIMER_USER` timer expires.

The `ITIMER_PROF` timer is used internally by the profiling system, and should not be used by applications.

The *itimerval.it_func* field is only valid for the `ITIMER_PROF` and `ITIMER_USER` timers. If non-null then the specified function is called directly at interrupt time, rather than sending a signal. The first argument passed to the function specifies the *delta* from the expected interrupt time (e.g. due to interrupt delays), and the second argument is the interrupt exception context (see Section 20.2.1 “C-level Exceptions”).

Three macros for manipulating time values are defined in <*sys/time.h*>; `timerclear()` sets a time value to zero; `timerisset()` tests if a time value is non-zero; and `timercmp()` compares two time values (beware that >= and <= do not work with this macro).

If the calls succeed, a value of 0 is returned. If an error occurs, the value -1 is returned.

For an example of how to use the asynchronous interval timing facilities, see the `com_itimer()` function in the example program #3, as described in Section 9.1.3 “Command Line Monitor (minimon)”.

19.2 PCI Bus Support

On boards that have a PCI bus, and have implemented the necessary machine-dependent low-level support code, a generic interface to the PCI bus is provided to handle bus initialization, enumeration and address mapping.

Below we describe these functions in detail, and an example of their use can be found in Section 9.1.11 "PCI Bus Demo". In all cases you will need to add the following include directives to your source file:

```
#include <pci/pcivar.h>
#include <pci/pcireg.h>
```

`void _pci_init (void)`
: Initialises the PCI bus controller and then scans the bus for devices, allocating address space for memory and i/o apertures and computing bus latency timers, etc; PCI-PCI bridges are also initialised and their buses scanned recursively. If running in RAM under control of a PROM monitor (e.g. PMON or IDT/sim), then the bus configuration is non-destructively scanned in order to determine the existing configuration. It is rarely necessary to call this function directly – it is called automatically at program initialization if any of the following PCI interface functions are used.

`pcitag_t _pci_find (const struct pci_match *matchp, unsigned int matchnum)`
: Scans the PCI bus for the *matchnum*'th device (starting at zero) which matches the ID and Class values in the structure pointed to be *matchp*:

```
struct pci_match {
    pcireg_t    class, classmask;
    pcireg_t    id, idmask;
}
```

A match succeeds when:

```
((device-class-reg & matchp->classmask) == matchp->class
 && (device-id-reg & matchp->idmask == matchp->id))
```

By using various combinations of mask value you can match all devices on the bus (`mask==id==0`), or all devices of a particular class and sub-class (e.g. `class==`*`mass-storage`* and `subclass==`*`ide`*), or a known manufacturer/device combination.

The function returns a PCI "tag" – a hardware-dependent token which represents the bus number, device number and sub-function number of the device's configuration space registers. It is passed to other functions below to gain access to other device registers and address spaces. When no matching devices are found, the function returns `~(pcitag_t)0`.

`void _pci_break_tag (pcitag_t tag, int *busp, int *devp, int *funcp)`
: Converts the hardware-dependent *tag* into the individual bus, device and function number. If any *busp*, *devp* or *funcp* are null pointers, then that value is not returned.

`void _pci_tagprintf (pcitag_t tag, const char *fmt, ...)`
: Calls the low-level `_mon_printf` function to print a diagnostic message, preceded by the string "`PCI bus` *`busno`* `slot` *`devno/funcno`*`:`".

`void _pci_devinfo (pcireg_t id, pcireg_t class, char *bufp, int *supp)`
: Returns a printable form of the manufacturer, device name and type in the buffer pointed to by `bufp`, keyed on a device's ID and CLASS config space registers. There is a large database of PCI devices, but it may not have yours!. The final parameter `supp` should always be NULL.

`pcireg_t _pci_conf_read32 (pcitag_t tag, int reg)`

`pcireg_t _pci_conf_read16 (pcitag_t tag, int reg)`

`pcireg_t _pci_conf_read8 (pcitag_t tag, int reg)`
: Returns the 32, 16 or 8 bit register at offset *reg* in the config space of the device selected by *tag*. If a master or target abort occurs then the value `0xffffffff` is returned, and the error is cleared.

`void _pci_conf_write32 (pcitag_t tag, int reg, pcireg_t val)`

`void _pci_conf_write16 (pcitag_t tag, int reg, pcireg_t val)`

`void _pci_conf_write8 (pcitag_t tag, int reg, pcireg_t val)`
Writes *val* to the 32, 16 or 8 bit register at offset *reg* in the config space of the device selected by *tag*.

`pcireg_t _pci_statusread (void)`
Returns the PCI host bridge's command/status register; this may be used to check for master or target aborts, and other error conditions.

`void _pci_statuswrite (pcireg_t stat)`
Writes *stat* to the PCI host bridge's command/status register; used to clear latched error signals.

`int _pci_map_mem (pcitag_t tag, int reg, vm_offset_t *vap, vm_offset_t *pap)`
Reads a PCI device's memory space base register (*reg* = 0x10 to 0x28 or 0x30) from the configuration space of the device selected by *tag*, and returns a CPU virtual address which will map to that PCI aperture in **vap*; the corresponding CPU physical address is returned in **pap*. Note that the physical PCI bus address stored in the device's base register may not correspond in a simple way to the CPU physical or virtual address. Returns 0 if all goes well, or -1 if the operation fails.

`int _pci_map_io (pcitag_t tag, int reg, vm_offset_t *vap, vm_offset_t *pap)`
Like `_pci_map_mem`, but maps an i/o space base register.

`int _pci_map_int (pcitag_t tag)`
Returns the ''interrupt number'' for the device selected by *tag*. The value returned is zero if the device does not have an interrupt line, and negative if there is a problem finding the corresponding interrupt number.

`vm_offset_t _pci_dmamap (vm_offset_t pa, unsigned int len)`
Maps the CPU physical address of a region of DRAM bounded by *pa* and *pa+len* to a PCI address, which can be passed to a PCI bus master device for ''DMA'' purposes. Note that there may be no direct correspondence between CPU and PCI addresses.

`vm_offset_t _pci_cpumap (vm_offset_t pcia, unsigned int len)`
Performs the reverse of the `_pci_dmamap` transformation , and converts a PCI memory address to a CPU physical address.

`void _pci_flush (void)`
Ensures that any software-visible PCI host bridge read-ahead fifos are empty.

`void _pci_wbflush (void)`
Ensures that any software-visible PCI host bridge write buffers are flushed to PCI.

`int _pci_cacheline_log2 (void)`
Returns log2() of the PCI cacheline size which should be programmed into any device which needs to know that value.

`int _pci_maxburst_log2 (void)`
Returns log2() of the maximum PCI burst length supported by the PCI host bridge.

`void * _isa_map_mem (vm_offset_t addr)`
Some legacy PCI devices (e.g. VGA cards) start up with fixed mappings in a virtual ISA memory bus (the bottom 16MB of PCI memory space). This function returns a CPU virtual address pointer which maps to address *addr* within the ISA memory space.

`void * _isa_map_io (unsigned int port)`
Similar to `_isa_map_mam`() but for access to the virtual ISA i/o bus (the bottom 1MB of PCI i/o space); returns the CPU virtual address which maps to ISA i/o port *port*.

`vm_offset_t _isa_dmamap (vm_offset_t pa, unsigned int len)`
Like `_pci_dmamap` but for ISA DMA devices.

`vm_offset_t _isa_cpumap (vm_offset_t isaa, unsigned int len)`
Like `_pci_cpumap` but for ISA DMA devices.

CPU Management

The second major component of the SDE run-time system consists of a set of support functions with which to initialise and maintain a MIPS architecture processor's caches, TLB and coprocessor registers; together with a powerful exception and interrupt handling mechanism, and support for remote source debugging of rommable code.

20.1 CPU Initialization

For rommable programs this code is invisible to your "application" program, as it is invoked automatically after a hardware reset, and before calling your `main()` function. It is described in detail in Section 21.4.1 "CPU Reset Handling".

20.2 Exception and Interrupt Handling

SDE has sample code – MTK customers get full sources – showing how to handle exceptions and interrupts in the MIPS architecture. The code supplied is certainly usable in a simple system.

The monitor-specific code hooks SDE's exception handling into the PROM monitor's own exception handling mechanism. This allows application programs to use the interface described here, whilst other exceptions (e.g. breakpoints) continue to be handled by the PROM monitor (e.g. the YAMON monitor).

20.2.1 C-level Exceptions

The run-time system provides a simple but powerful exception handling mechanism called *xcptions*, which are modelled on the POSIX *signal* handling mechanism described in Section 19.1.6 "Signal Handling". To use it include the header file *<mips/xcpt.h>* which defines these interfaces:

```
typedef int (*xcpt_t)(int, struct xcptcontext *);

struct xcptaction {
        xcpt_t          xa_handler;
        unsigned        xa_flags;       /* unused */
};

/* install xcption handler */
int     xcptaction (int xcptno, struct xcptaction *act,
                        struct xcptaction *oact);

/* install xcption handler (simple version) */
xcpt_t  xcption (int xcptno, xcpt_t handler);
```

The `xcptaction()` function is similar to the POSIX `sigaction()` function. If `act` is non-zero, then it specifies a handler routine to be called when exception `xcptno` occurs (as defined in *<mips/xcpt.h>*). If `oact` is non-zero, then the previous handling information for that exception is returned to the caller. The function returns zero on success, or a non-zero error code if the parameters are faulty.

Once a handler is installed, it remains installed until another `xcptaction()` call is made for the same exception number. Note that the `xcptaction.xa_flags` field is currently ignored, but is intended to allow control over which registers are saved and how the exception is vectored; it should be set to zero.

The `xcption()` function provides a simpler interface, analogous to the old UNIX *signal* function. It is passed a simple function pointer, or **`XCPT_DFL`** to restore the default handler. It returns a pointer to the previous handler function, or **`XCPT_ERR`** on error.

When an exception occurs the appropriate *xcption* handler is called with two arguments:

1) the exception number;
2) a pointer to the *xcptcontext* structure which holds the processor state at the time of the exception, for example:

```
int handler (int xcptno, struct xcptcontext *xcp)
```

The *xcption* handler should normally return 0.

For an example showing the use of *xcptions*, see Section 9.1.2 "TLB Exception Handling (tlbxcpt)".

Error handling

As stated above, an *xcption* handler should normally return 0. But if it cannot handle the exception properly, or needs to asynchronously inform the application of some event, then it can return a non-zero POSIX *signal number*, as defined in Section 19.1.6 "Signal Handling". The run-time system contains a default exception handler, which simply translates MIPS exception numbers into the appropriate POSIX signal numbers.

The application's signal handler, if installed by `sigaction()` or `signal()`, will be called before returning to the interrupted/failing instruction; if the signal handler then returns normally, execution will continue with the interrupted instruction. If no signal handler is installed, then the application will instead be terminated with a diagnostic message showing the cause of the exception, a register dump, and a stack trace. Note that SIGKILL cannot be caught, so it is guaranteed to terminate the application.

If your application has been built to run on an MDI target (e.g. the MIPSsim simulator or a CPU connected by an EJTAG probe), or it includes the SDE remote debug stub (see Section 21.4.3 "Remote Debug Stub"), then *gdb* will be activated whenever any exception handler returns a non-zero result, just before it gets passed to the application's signal handler. This lets you use *gdb* to analyse exceptional events. But when you are using a PROM monitor's remote debug facilities (e.g. YAMON), then only "uncaught" exceptions will be seen by *gdb*: if you've installed an SDE exception handler then that exception will not be reported to *gdb*, whatever its result, unless you set a breakpoint in the exception handler itself, or in the `xcpt_default` function.

```
/* diagnostics */
void    xcpt_show (struct xcptcontext *xcp);
void    xcptstacktrace (struct xcptcontext *xcp);
```

An exception handler may call `xcpt_show()` and/or `xcptstacktrace()` explicitly, to display diagnostic messages without terminating the application.

Note that all interrupts are disabled during exception processing, unless they are explicitly unmasked inside your *xcption* or *intrupt* handler.

20.2.2 RTOS Context Switch

RTOS developers and porters may find the following functions useful.

```
/* return to different xception context */
void    xcptrestore (struct xcptcontext *xcp);

/* low-level setjmp/longjmp */
int     xcptsetjmp (xcptjmp_buf *xjb);
void    xcptlongjmp (xcptjmp_buf *xjb, int val);
```

The `xcptsetjmp()` and `xcptlongjmp()` functions are analogous to the standard C library `setjmp` and `longjmp` functions, but rather than saving and restoring the high-level POSIX signal mask, they save and restore the MIPS coprocessor 0 *Status* register (i.e. the interrupt mask), along with the stack pointer, program counter, and the other *callee−saved* registers. These functions can be used to implement a context save/restore for threads that have voluntarily blocked (e.g. due to a locked semaphore).

The `xcptrestore()` function allows an explicit return to a different xception context, i.e. not the one that you are currently servicing. This can be used to implement a context switch to a thread that has been scheduled by an external event (i.e. an interrupt).

Since it is unlikely that multiple threads will be using the floating point unit simultaneously, we recommend that the floating point context switch should be lazy: enable the *Status.CU1* bit only for the current FPU owner, and then switch the FPU registers only upon receiving a *CoProcessor Unusable* (**`XCPTCPU`**) exception.

20.2.3 C-level Interrupts

On almost all MIPS processors there are 8 level-sensitive interrupt "inputs" (6 hardware and 2 software). If any become active, and they are enabled by the mask bits in the CPU's *Status* register, then the processor generates an Interrupt (**XCPTINTR**) exception. Software must then examine the pending bits in the *Cause* register to determine which of the 8 interrupts are active, prioritise them and then vector to the relevant interrupt handler.

We provide a mechanism called *intrupts* to handle this: it is very similar to the *xcption* mechanism described above, but with an additional interrupt prioritization scheme. Of course *intrupts* are just a special class of *xcption*, and is defined in header file <*mips/xcpt.h*>.

```
struct intraction {
        xcpt_t          ia_handler;     /* interrupt handler function */
        int             ia_arg;         /* passed to interrupt handler */
        unsigned        ia_ipl;         /* priority (1-8, 0=off) */
};

/* install intrupt handler */
int     intraction (unsigned int intrno, struct intraction *act,
                        struct intraction *old);

/* install intrupt handler (simple version) */
xcpt_t  intrupt (unsigned int intrno, xcpt_t handler, int arg);
```

The `intraction()` function installs an *intrupt* handler, just like `xcptaction()` described above. The `intrno` argument is a number in the range 0 to 7, specifying which interrupt-pending bit in the *Cause* register this action refers to. The `intraction.ia_arg` field specifies an arbitrary value to be passed to the *intrupt* handler, which might be used to allow a common handler to distinguish between two distinct devices.

The `intrupt()` function provides a simpler way to install an interrupt handler. It is like the `xcption()` function described above, but its `arg` parameter fulfills the same task as the `intraction.ia_arg` field.

When an interrupt occurs the appropriate *intrupt* handler is called with two arguments:

1) the `ia_arg` parameter;

2) a pointer to the *xcptcontext* structure which holds the processor state at the time of the interrupt, for example:

```
int handler (int arg, struct xcptcontext *xcp)
```

Like an *xcption* handler, an *intrupt* handler should normally return 0, but can return a *signal* number if it wants to send an asynchronous signal to the application. For instance a "debug button" interrupt handler could return **SIGTRAP** to enter the debugger.

Some boards may multiplex several interrupts onto each CPU interrupt line, and they will require a second level interrupt handler that uses an external interrupt request register to select the correct interrupt function.

Warning: interrupt handlers should not expect to be able to safely change the *Status* register saved in `xcp->sr` if the non-interrupt code itself modifies the Status register non-atomically (e.g. using `mips_bissr()`, `spl()`, etc). Coprocessor register updates can never be atomic[20], and there is no simple way to serialise access to the *Status* register. Contact us for advice if you need to do this.

For an example program showing the use of *intrupts*, see Section 9.1.14 "Interrupt Example".

Interrupt Priorities

Remember that until very recently MIPS processors have not supported hardware interrupt prioritization, and it has traditionally been up to software to implement whatever priority scheme it requires. Our *intrupt* mechanism implements a fixed-priority software-based scheme, whereby each interrupt input can be assigned to one of 8 fixed *interrupt priority levels* (IPLs). This is not a one-to-one mapping: any number of interrupt inputs can be assigned

[20] Actually the MIPS32 Release 2 ISA does allow atomic changes of the CP0 *Status* register, but only to the interrupt enable (IE) bit.

the same IPL, and in any combination.

The `intraction.ia_ipl` field, passed to the `intraction()` function, explicitly specifies that interrupt's IPL. But the simpler `intrupt()` function uses a default IPL derived from the interrupt number as follows:

Table 20-1 Interrupt priorities

Input	Cause Reg	IPL
h/w interrupt 5	IP7	8 <-HIGHEST
h/w interrupt 4	IP6	7
h/w interrupt 3	IP5	6
h/w interrupt 2	IP4	5
h/w interrupt 1	IP3	4
h/w interrupt 0	IP2	3
s/w interrupt 1	IP1	2
s/w interrupt 0	IP0	1
		0 <-LOWEST

In this model the CPU is notionally set to a priority level between 0 and 8 (inclusive): being set to a given priority level means that all interrupts at that IPL and below are masked out, and all above are enabled. Thus if the CPU is at priority level 0 it means that all interrupts are enabled, and if at level 8 then all are disabled. Normally your application will be running at level 0.

When an interrupt handler is called, the CPU priority is automatically set to that interrupt's IPL for the duration of the call to the handler. This prevents nested interrupts from the same device, or lower-priority devices, but allows them from higher priority devices.

Device drivers and other code will sometimes need to explicitly block out some or all interrupts in critical regions. This is done by temporarily ''raising'' and then ''lowering'' the CPU's priority level, using these functions.

```
unsigned int spl (unsigned int ipl);
unsigned int splx (unsigned int x);
```

Here `spl()` sets the CPU's priority level to `ipl`, and returns a value that can be passed later to `splx()`, to restore the old priority. Note that this return value is opaque: it is not the old priority level. This leads to the following typical usage:

```
{
    unsigned int s = spl (5);   /* block out level 5 i/us and below */
    /* CRITICAL REGION HERE */
    (void) splx (s);            /* return to previous priority level */
}
```

For very short critical sections only it may be faster to disable all interrupts:

```
{
    unsigned int s = _mips_intdisable ();
    /* CRITICAL REGION HERE */
    _mips_intrestore (s);
}
```

You can test for a pending interrupt while it is blocked, using

```
int     intrpending (unsigned int intrno);
```

which returns 1 if CPU h/w interrupt pending bit `intrno` is active.

Software interrupts

The MIPS *Cause* register includes two *software interrupt* bits, which allow high-priority interrupt handlers to request a new interrupt at a low-priority, or non-interrupt code to kick-start interrupt-level processing. The following functions provide a safe way to switch these interrupts on and off.

```
void    siron (unsigned int intrno);
void    siroff (unsigned int intrno);
```

Note that `intrno` may only be 0 or 1, and the respective interrupt handlers must call `siroff()` to remove the interrupt request before they return.

20.3 Cache Maintenance

The cache management function prototypes are supplied by including <*mips/cpu.h*>. Many of these routines expect to be passed an address range to operate on, consisting of a starting *virtual address*, and a byte count.

`void mips_size_cache (void)`
Size the caches, setting the following global variables:

- *mips_icache_size, mips_icache_linesize, mips_icache_ways* : The size (in bytes) of the primary instruction cache; the size of each cache line, and the number of ways of set associativity.
- *mips_dcache_size, mips_dcache_linesize, mips_dcache_ways* : Ditto for the primary data cache.
- *mips_scache_size, mips_scache_linesize, mips_scache_ways* : Ditto for the secondary cache, if present.

`void mips_init_cache (void)`
Size the caches as above, and initialise them. The function MUST be called after a hardware reset and before using the caches, otherwise they may be in an inconsistent state. This is normally called by the standard reset code. Do NOT call it from application code, as it may invalidate dirty cache lines in a writeback cache, without actually writing them back to memory.

`void mips_sync_icache (vaddr_t va, size_t n)`
Synchronises the icache with the dcache, which is necessary when the instruction stream is modified by software (e.g. inserting software breakpoints, self-modifying code, etc).

`void mips_clean_cache (vaddr_t va, size_t n)`
Write back and invalidate entries matching the given address range from all caches. The most common routine to call in device drivers before starting a DMA transfer, or after dynamically modifying executable code.

`void mips_clean_dcache (vaddr_t va, size_t n)`
Write back and invalidate entries matching the given address range from the data caches only – separate instruction caches are unchanged.

`void mips_clean_icache (vaddr_t va, size_t n)`
Invalidate entries matching the given address range from the instruction caches only – separate data caches are unchanged.

`void mips_flush_cache (void)`
Write back and invalidate all entries from all caches. The simplest way to completely synchronise caches and memory, but not necessarily the most efficient.

`void mips_flush_dcache (void)`
Write back and invalidate all entries from all data caches – separate instruction caches are unchanged.

`void mips_flush_icache (void)`
Invalidate all entries from all instruction caches – separate data caches are unchanged.

`void mips_lock_icache (vaddr_t va, size_t n)`

`void mips_lock_dcache (vaddr_t va, size_t n)`

`void mips_lock_scache (vaddr_t va, size_t n)`
On CPUs which support cache locking, these functions allow you to lock regions of code or data into the primary instruction, data or secondary caches respectively. Take care not to use the global *flush* functions after

locking caches, as they will invalidate (and unlock) the locked cache lines.

20.4 TLB Maintenance

The functions listed below provide for initialization and maintenance of the CPU's memory management *Translation Lookaside Buffer* (TLB), if present. An example showing the use of these functions can be found in Section 9.1.2 "TLB Exception Handling (tlbxcpt)". The TLB and memory management definitions are supplied by including <*mips/cpu.h*>.

`void mips_init_tlb (void)`
Initialises and invalidates the whole TLB.

`unsigned int mips_tlb_size (void)`
Returns the number of entries in the TLB.

`void mips_tlbinval (tlbhi_t hi)`
Probes the TLB for an entry matching `hi`, and if present invalidates it.

`void mips_tlbinvalall (void)`
Invalidate the whole TLB.

`void mips_tlbri2 (tlbhi_t *phi, tlblo_t *plo0, tlblo_t *plo1, unsigned *pmsk, int index)`
Reads the TLB entry with specified by `index`, and returns the *EntryHi*, *EntryLo0*, *EntryLo1* and *PageMask* parts in `*phi`, `*plo0`, `*plo1` and `*pmsk` respectively.

`void mips_tlbwi2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk, int index)`
Writes `hi`, `lo0`, `lo1` and `msk` into the TLB entry specified by `index`.

`void mips_tlbwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk)`
Writes `hi`, `lo0`, `lo1` and `msk` into the TLB entry specified by the *Random* register.

`int  mips_tlbprobe2 (tlbhi_t hi, tlblo_t *plo0, tlblo_t *plo1, unsigned *pmsk)`
Probes the TLB for an entry matching `hi` and returns its index, or -1 if not found. If found, then the *EntryLo0*, *EntryLo1* and *PageMask* parts of the entry are also returned in `*plo0`, `*plo1` and `*pmsk` respectively.

`int  mips_tlbrwr2 (tlbhi_t hi, tlblo_t lo0, tlblo_t lo1, unsigned msk)`
Probes the TLB for an entry matching `hi` and if present rewrites that entry, otherwise updates a random entry. A safe way to update the TLB.

20.5 Hardware Watchpoints

Some MIPS architecture CPUs provide one or more hardware watchpoint registers in Coprocessor 0 (these are separate from any EJTAG hardware breakpoint registers). The watchpoint registers generate a CPU exception when software loads or stores data, or executes instructions, within a programmable address range. Different MIPS-Based CPUs implement very different watchpoint controls (number of watchpoints, type of access, physical/virtual address, address masking, and so on). To make this manageable and portable between different CPUS we have developed a generic API which is documented here. These facilities are used by the SDE remote debug stub to support *gdb*'s watchpoint facility; but you could also use them to implement profiling or debugging facilities within your own software.

To use the watchpoint API described here you include the file <*mips/watchpoint.h*>.

`int _mips_watchpoint_init (void)`
Initialises the watchpoint system and returns the number of hardware watchpoints available.

`int _mips_watchpoint_howmany (void)`
Just returns the number of hardware watchpoints, without reinitialsing the sub-system.

`int _mips_watchpoint_capabilities (int wpnum)`
Returns the *capability* of watchpoint number *wpnum* (0 to n). Usually called after `_mips_watchpoint_init()` to collect and cache each watchpoint's capability. The capability is the bitwise OR of some or all of the following values:

Table 20-2 Hardware watchpoint attributes

MIPS_WATCHPOINT_SSTEP	Hardware single-step supported.
MIPS_WATCHPOINT_VALUE	Can qualify the watchpoint with the value of the data being read or written from/to memory.
MIPS_WATCHPOINT_ASID	Can qualify match using the virtual address-space ID (ASID).
MIPS_WATCHPOINT_VADDR	Matches against virtual address (if not set then matches against physical address).
MIPS_WATCHPOINT_RANGE	Supports an address range (arbitrarily aligned start and end address).
MIPS_WATCHPOINT_MASK	Supports an address mask (size must be a power-of-two, and start address aligned on a matching boundary).
MIPS_WATCHPOINT_DWORD	Only supports an address match within a single 8 byte aligned double word; if an address range/mask is supported then the minimum size and aligment is 8 bytes.
MIPS_WATCHPOINT_WORD	Only supports an address match within a single 4 byte aligned word; if an address range/mask is supported then the minimum size and aligment is 4 bytes.
MIPS_WATCHPOINT_X	Instruction fetch breakpoint supported.
MIPS_WATCHPOINT_R	Data read breakpoint supported.
MIPS_WATCHPOINT_W	Data write breakpoint supported.

```
int _mips_watchpoint_set (int type, int asid, vaddr_t va,
                          paddr_t pa, size_t size)
```

Creates a new watchpoint where: *type* is the OR of the last three capabilities (i.e. instruction fetch, read and/or write); *asid* is the virtual address space ID (or -1 for global); *va* is the virtual address of the start of the watchpoint region; *pa* is the physical address (can be zero if virtual address matching is supported); and *size* is the size of the watchpoint region.

For CPUs which support an address mask, *addr* and *size* can be arbitrarily aligned, and the code will compute the smallest aligned region which fits around them. Beware that this could get quite loose, and cause a large number of false watchpoint hits.

The return value indicates the success or failure as follows:

Table 20-3 Watchpoint return codes

MIPS_WP_OK	Succeeded.
MIPS_WP_NOTSUP	This type of watchpoint is not supported, or possibly you've asked for a watchpoint region which is larger than can be supported.
MIPS_WP_INUSE	All hardware resources which support this type of watchpoint are in use.
MIPS_WP_NOMATCH	Matching watchpoint cannot be found (see `_mips_watchpoint_clear()` below).
MIPS_WP_OVERLAP	Address range would overlap the debugger's own code, data or stack.
MIPS_WP_BADADDR	If the *pa* value is zero and virtual address matching is not supported.

```
int _mips_watchpoint_clear (int type, int asid, vaddr_t va, size_t size)
```

Delete a watchpoint: the parameters must match those used when the watchpoint was created by `_mips_watchpoint_set()`. See `_mips_watchpoint_set()` for the return codes.

```
int _mips_watchpoint_set_callback (int asid, vaddr_t va, size_t len)
```

A callback function which you can (optionally) provide. When a new watchpoint is about to be added, your code has a last chance to check the computed address range to make sure that it doesn't overlap with its own code or data (which could cause recursive watchpoint traps). Should return MIPS_WP_OK or MIPS_WP_OVERLAP. If you don't provide this function then all watchpoints are allowed.

```
int _mips_watchpoint_hit (const struct xcptcontext *xcp,
```

```
                                vaddr_t *vap, size_t *sizep)
```

Called by your hardware watchpoint exception handler (usually the debug stub) to check whether the exception context *xcp* was a true watchpoint hit. If so the return value will be non-zero, and contain one of MIPS_WATCHPOINT_R, MIPS_WATCHPOINT_W or MIPS_WATCHPOINT_X to indicate the type of access. If in addition the bit MIPS_WATCHPOINT_INEXACT is set then this was a watchpoint exception, but it was based on a loose address mask, and this access was outside of the range originally requested by `_mips_watchpoint_set()`; your code must single-step over this instruction and then continue.

`void _mips_watchpoint_remove (void)`

Called by the debug stub, or your watchpoint exception handler, to disable hardware watchpoints, e.g. before single-stepping over an instruction which may trigger the watchpoint.

`void _mips_watchpoint_insert (void)`

Called by the debug stub, or watchpoint exception handler, to enable hardware watchpoints, e.g. after single-stepping over an instruction and before continuing execution.

`void _mips_watchpoint_reset (void)`

Clear all watchpoints.

20.6 System Coprocessor (CP0) Intrinsics

All MIPS-Based CPUs contain a "System Control" subsystem known as Coprocessor 0, or CP0. This is used by operating systems and other low-level software to control interrupts, exceptions, memory management, caches, etc. These *intrinsics* provide very low-level access to the CP0 registers from C and C++ code. Other intrinsics which give access to "user-level" instructions and registers are described in a separate chapter, see Chapter 18 "Intrinsics for MIPS® Architecture".

The header file <*mips/cpu.h*> (which in turn includes the appropriate cpu-specific header), defines the following intrinsics:

For each of the register access intrinsics listed below, the "`*`" symbol represents up to five separate intrinsics, as follows:

Table 20-4 CP0 register access intrinsics

`*`	*Arguments*	*Operation*
`get`	`()`	Returns the register value.
`set`	`(unsigned val)`	Sets the register to `val`, and returns void.
`xch`	`(unsigned val)`	Sets the register to `val`, and returns the **old** register value.
`bis`	`(unsigned set)`	Bit set (*`reg |= set`*): returns the **old** register value. Only defined for registers with bit-fields.
`bic`	`(unsigned clr)`	Bit clear (*`reg &= ~clr`*): returns the **old** register value. Only defined for registers with bit-fields.
`bcs`	`(unsigned clr, unsigned set)`	Bit clear and set (*`reg = (reg & ~clr)	set`*): returns the **old** register value. Only defined for registers with bit-fields.

Common CP0 Registers

Some of the CP0 registers are common between almost all MIPS-Based CPU families, and the intrinsics to access these have the common prefix `mips_`.

Remember though that even for the common registers, the internal bit definitions are not necessarily the same across all CPU types. Make sure that you include the generic <*mips/cpu.h*>, and not <*mips/m32c0.h*>, or any of the CPU-specific header files.

N.B. The intrinsics which manipulate the coprocessor registers do not provide atomicity in the presence of interrupts or other exceptions. This can be particularly important if you are changing the *Cause* or *Status* registers. If possible, avoid read-modify-write operations on the *Status* register: write only constant values, or stored values manipulated only by atomic operations, unless you know that interrupts are already disabled (e.g. because you're in an exception handler). Ensure that interrupts are disabled when you update the *Cause* register.

`mips_*sr`
: (i.e. *mips_getsr*, *mips_setsr*, *mips_xchsr*, *mips_bissr*, *mips_bicsr*). Operations on the *Status* register (CP0 register 12). See the atomicity warning above.

`mips_*cr`
: Operations on the *Cause* register (CP0 register 13). See warning above.

`mips_getcount, mips_setcount`

`mips_getcompare, mips_setcompare`
: Operations on the *Count* and *Compare* registers (CP0 registers 9 and 11). Available on most modern MIPS architecure CPUs, these implement an on-chip timer.

`mips_getprid`
: Return the read-only *PrID* register (CP0 register 15). See <*mips/prid.h*> for a list of known values.

`mips_*config`
Operations on *Config* register (CP0 register number varies).

`mips_*ecc`
Operations on *ECC* register (CP0 register 26), used for cache error correction on some MIPS III+ CPUs.

`mips_*context`
Operations on the *Context* register (CP0 register 4).

`mips_*pagemask`
Operations on the *PageMask* register (CP0 register 5).

`mips_*wired`
Operations on the *Wired* register (CP0 register 6).

`mips_*entrylo`
Operations on the *EntryLo* register (CP0 register 2).

`mips_*entryhi`
Operations on the *EntryHi* register (CP0 register 10).

`mips_*taglo`

`mips_*taghi`
Operations on *TagLo* and *TagHi* registers (CP0 registers 28 and 29), used for cache testing and maintenance on many MIPS architecture CPUs.

`mips_*watchlo`

`mips_*watchhi`
Operations on *WatchLo* and *WatchHi* registers (CP0 registers 18 and 19), used for hardware watchpoints on many MIPS III+ CPUs.

CP0 Registers of MIPS32®/MIPS64® Architecture

The include files <*mips/m32c0.h*> and <*mips/m32tlb.h*> defines the coprocessor registers and memory-management unit of CPUs conforming to the MIPS32/MIPS64 specifications. They include the following functions:

`mips32_*config0`
Operations on the *Config0* register (CP0 register 16, select 0), also available via the generic `mips_*config` functions described above.

`mips32_getconfig1`
Returns the *Config1* register (CP0 register 16, select 1).

`mips32_getconfig2`
Returns the *Config2* register (CP0 register 16, select 2).

`mips32_getconfig3`
Returns the *Config3* register (CP0 register 16, select 3).

`mips32_getwatchlo(int sel)`
Return the *WatchLo* register numbered *`sel`*.

`mips32_setwatchlo(int sel, unsigned int val)`
Set the *WatchLo* register numbered *`sel`* to *`val`*.

`mips32_getwatchhi(int sel)`
Return the *WatchHi* register numbered *`sel`*.

`mips32_setwatchhi(int sel, unsigned int val)`
Set the *WatchHi* register numbered *`sel`* to *`val`*.

`mips32_*errctl`
Operations on the *ErrCtl* register (CP0 register 26, select 0).

`mips32_*datalo`
Operations on the *DataLo* register (CP0 register 28, select 1).

CP0 Registers of MIPS32®/MIPS64® Release 2 Architecture

The MIPS32 Release 2 ISA defines a few new Coprocessor 0 registers, also defined in include files *<mips/m32c0.h>*.

`mips32_*pagegrain`
: Operations on the MIPS32 Release 2 *PageGrain* register (CP0 register 5, select 1).

`mips32_*hwrena`
: Operations on the MIPS32 Release 2 *HWREna* register (CP0 register 7, select 0).

`mips32_*intctl`
: Operations on the MIPS32 Release 2 *IntCtl* register (CP0 register 12, select 1).

`mips32_*srsctl`
: Operations on the MIPS32 Release 2 *SRSCtl* register (CP0 register 12, select 2).

`mips32_*srsmap`
: Operations on the MIPS32 Release 2 *SRSMap* register (CP0 register 12, select 3).

`mips32_*ebase`
: Operations on the MIPS32 Release 2 *EBase* register (CP0 register 15, select 1).

Shadow Sets of MIPS32®/MIPS64® Release 2 Architecture

The MIPS32 Release 2 architecture adds support for alternative ‘‘shadow’’ banks of CPU general purpose registers, for use by low-latency interrupt and exception handlers. These intrinsics allow C code to read and write registers in other shadow sets, and are defined in include files *<mips/m32c0.h>*.

`uint32_t _mips32r2_xchsrspss(uint32_t set)`
: Sets the *PSS* field in the *SRSCtl* register to `set`, allowing access to that shadow set with the following intrinsics. Returns the old value of the *PSS* field.

`uint32_t _mips32r2_rdpgpr(int regno)`
: Returns register number `regno` from the selected shadow set. The `regno` argument must be a constant between 0 and 31.

`void _mips32r2_wrpgpr(int regno, uint32_t val)`
: Sets register number `regno` in the selected shadow set to `val`. The `regno` argument must be a constant between 0 and 31.

CP0 Registers of MIPS® MT ASE

The include file *<mips/mt.h>* defines the coprocessor registers introduced by the MT ASE, and includes the following C access functions:

`mips32_*mvpcontrol`
: Operations on the MVPControl Register (CP0 Register 0, Select 1).

`mips32_*mvpconf0`
: Operations on the MVPConf0 Register (CP0 Register 0, Select 2).

`mips32_*mvpconf1`
: Operations on the MVPConf1 Register (CP0 Register 0, Select 3).

`mips32_*vpecontrol`
: Operations on the VPEControl Register (CP0 Register 1, Select 1).

`mips32_*vpeconf0`
: Operations on the VPEConf0 Register (CP0 Register 1, Select 2).

`mips32_*vpeconf1`
: Operations on the VPEConf1 Register (CP0 Register 1, Select 3).

`mips32_*yqmask`
: Operations on the YQMask Register (CP0 Register 1, Select 4).

`mips32_*vpeschedule`
: Operations on the VPESchedule Register (CP0 Register 1, Select 5).

`mips32_*vpeschefback`
: Operations on the VPEScheFback Register (CP0 Register 1, Select 7).

`mips32_*tcstatus`
: Operations on the TCStatus Register (CP0 Register 4, Select 1).

`mips32_*tcpc`
: Operations on the TCPC Register (CP0 Register 4, Select 2).

`mips32_*tchalt`
: Operations on the TCHalt Register (CP0 Register 4, Select 3).

`mips32_*tccontext`
: Operations on the TCContext Register (CP0 Register 4, Select 4).

`mips32_*tcschedule`
: Operations on the TCSchedule Register (CP0 Register 4, Select 5).

`mips32_*tcschefback`
: Operations on the TCScheFback Register (CP0 Register 4, Select 6).

`mips32_*srsconf*`
: Operations on the SRSConf0-4 Registers (CP0 Register 6, Select 1-5)

The MT ASE also permits access to registers with a different thread context or virtual processor.

`mips32_mt_settarget (int vpe, int tc)`
: Selects the target VPE and TC number for the following access functions.

`mips32_mt_getc0status()`
: Return the CP0 Status register of the selected TC/VPE.

`mips32_mt_setc0status(int val)`
: Set the CP0 Status register of the selected TC/VPE.

`mips32_mt_getc0cause()`
: Return the CP0 Cause register of the selected TC/VPE.

`mips32_mt_setc0cause(val)`
: Set the CP0 Cause register of the selected TC/VPE.

`mips32_mt_getc0config()`
: Return the CP0 Config register of the selected TC/VPE.

`mips32_mt_setc0config(val)`
: Set the CP0 Config register of the selected TC/VPE.

`mips32_mt_getc0config1()`
: Return the CP0 Config1 register of the selected TC/VPE.

`mips32_mt_setc0config1(val)`
: Set the CP0 Config1 register of the selected TC/VPE.

`mips32_mt_getc0ebase()`
: Return the CP0 EBase register of the selected TC/VPE.

`mips32_mt_setc0ebase(val)`
: Set the CP0 EBase register of the selected TC/VPE.

`mips32_mt_getsp()`
: Return the stack pointer (`$29`) of the selected TC/VPE.

`mips32_mt_setsp(val)`
: Set the stack pointer (`$29`) of the selected TC/VPE.

`mips32_mt_getgp()`
Return the global pointer (`$28`) of the selected TC/VPE.

`mips32_mt_setgp(val)`
Set the global pointer (`$28`) of the selected TC/VPE.

`mips32_mt_getvpecontrol()`
Return the CP0 VPEControl register of the selected TC/VPE.

`mips32_mt_setvpecontrol(val)`
Set the CP0 VPEControl register of the selected TC/VPE.

`mips32_mt_getvpeconf0()`
Return the CP0 VPEConf0 register of the selected TC/VPE.

`mips32_mt_setvpeconf0(val)`
Set the CP0 VPEConf0 register of the selected TC/VPE.

`mips32_mt_gettcstatus()`
Return the CP0 TCStatus register of the selected TC/VPE.

`mips32_mt_settcstatus(val)`
Set the CP0 TCStatus register of the selected TC/VPE.

`mips32_mt_gettcbind()`
Return the CP0 TCBind register of the selected TC/VPE.

`mips32_mt_settcbind(val)`
Set the CP0 TCBind register of the selected TC/VPE.

`mips32_mt_gettcrestart()`
Return the CP0 TCRestart register of the selected TC/VPE.

`mips32_mt_settcrestart(val)`
Set the CP0 TCRestart register of the selected TC/VPE.

`mips32_mt_settchalt(val)`
Set the CP0 TCHalt register of the selected TC/VPE.

`mips32_mt_gettccontext()`
Return the CP0 TCContext register of the selected TC/VPE.

`mips32_mt_settccontext(val)`
Set the CP0 TCContext register of the selected TC/VPE.

20.7 Miscellaneous System Support

The following generic MIPS system support functions are defined in include file <*mips/cpu.h*>.

`void mips_wbflush (void)`
Drain the write buffer. All stores issued prior to the call are guaranteed to have been written to memory or device by the time the function returns. It should be called between writing to device control registers and reading their status/data registers. On some CPUs it is also necessary to call it between successive writes to the same register, to prevent word-gathering write-buffers from swallowing some of the writes.

`void _mips_sync (void)`
On modern MIPS-Based CPUs this generates a `sync` instruction. This is almost but not quite the same as `mips_wbflush()` – it is a memory *barrier* which guarantees that all memory accesses preceding this instruction will be completed before any accesses which follow this instruction. It says nothing though about external state, such as interrupts – and on simpler CPUs with blocking loads it may be interpreted as a no-op.

```
uint8_t  mips_get_byte (void *addr, int *err)
uint16_t mips_get_half (void *addr, int *err)
uint32_t mips_get_word (void *addr, int *err)
```

`uint64_t mips_get_dword (void *addr, int *err)`
Return the byte, halfword, word, or dword at address `addr`. If the address is invalid, then `*err` may be set to a non-zero value, otherwise `*err` is unchanged. You can use these functions when accessing arbitrary memory locations outside of your program, to ensure that peculiarities of your system or CPU address map are handled correctly.

`int mips_put_byte (void *addr, uint8_t val)`

`int mips_put_half (void *addr, uint16_t val)`

`int mips_put_word (void *addr, uint32_t val)`

`int mips_put_dword (void *addr, uint64_t val)`
Store a byte, halfword, word, or dword `val` to arbitrary address `addr`. If the address is invalid, then a non-zero value may be returned, otherwise they return zero.

20.8 Floating Point Coprocessor (CP1)

The generic header file *<mips/fpa.h>* defines constants and functions for controlling the floating point coprocessor (CP1) and its register set.

`int fpa_enable (int fast)`
Probes to see if CP1 is present. If so it is initialised, CP1 instructions are enabled, and 1 is returned. If it is not present, then CP1 instructions are disabled, and 0 is returned. If `fast` is non-zero then, if possible, the FPU is set to "performance mode" where IEEE−754 traps will not be taken for denormalised values, which will instead be flushed or rounded.

`void fpa_save (struct fpactx *ctx)`
Save all the floating point data registers and coprocessor state into the structure pointed to by *`ctx`*.

`void fpa_restore (const struct fpactx *ctx)`
Restore all the registers and coprocessor state from the structure pointed to by *`ctx`*.

`unsigned fpa_getrid (void)`
Returns on CP1 control register 0, the read-only floating point *RevisionID* register.

`fpa_*sr`
Operations on CP1 control register 31, the floating point control and status register. See Section 20.6 "System Coprocessor (CP0) Intrinsics" for a description of '`*`'.

20.8.1 Coprocessor 1 Emulation

The run-time system includes a complete MIPS coprocessor 1 (floating point) instruction emulator. It can emulate all floating point instructions when there is no hardware FPU, or just those instructions with operands that the FPU cannot handle (e.g. denormalised values, underflow, etc). The only public interface to the module is:

```
void    _cop1_init (int emulateall);
```

This function installs the appropriate exception or interrupt handler: a non-zero value for `emulateall` installs full emulation via the *CoProcessor Unusable* (**`XCPTCPU`**) exception, whilst a zero value installs only the floating point interrupt handler (or **`XCPTFPE`** exception handler on an R4000 CPU and above). You'll probably never need to call it yourself – it is normally invoked automatically by the standard run-time startup code, see Section 21.1.1 "Run-time Initialization".

A faster alternative to trap-based coprocessor emulation is to use the compiler's **–msoft–float** option, see Section 12.5 "Software Floating Point".

Embedded System Kit Source

This chapter introduces the source files which make up the embedded system kit. The directory `.../sde/kit` contains a collection of C source, assembler source and pre-compiled object files which fulfill two separate functions:

1) They form a run-time i/o system and environment for application programs, such as the examples. The programming interface provided by this system is described in Section 19.1 "POSIX API Environment".

2) They include a set of low-level primitives to initialise and manage a MIPS-Based CPU's caches, TLB, FPU, exceptions, interrupts, etc. The programming interface provided by these components is described in Chapter 20 "CPU Management".

The kit is set up so that you can build software, modelled on one of SDE's example programs, and by some judicious values for *makefile* variables, get the software to build successfully for any of a large number of different boards.

Unless you are using SDE *lite* then this is all supplied as source code, and can be adapted to other run-time environments, or perhaps just used for inspiration when porting a PROM monitor or operating system to the MIPS architecture.

The kit is built around the idea that each target has its own directory of software, and its own makefile; in the target makefile the ROM monitor (if any) and CPU type are identified, along with other options.

But first a note on the run-time i/o system.

21.1 POSIX System Interface

The run-time i/o system is modelled on the POSIX.1 specification (see [POSIX88]). It is implemented by the following files in `.../sde/kit/share`; they will be either C or assembler-with-cpp (*.sx*) files for supported SDE customers, or pre-compiled object files for other users:

- *crt0* : generic C/C++ run-time system startup code, see below.
- *env* : the `getenv`/`setenv` functions, which interface to a board-specific non-volatile environment variable store if present.
- *flashenv.c* and *flashrom.c* : support code for a simple NAME=VALUE environment store in FLASH.
- *flashdev* : implements the `/dev/flash` special device file, described in Section 19.1.4 "Flash Memory Device (/dev/flash)".
- *mfs* : implements a pseudo "memory file system" whose structure is defined by a monitor-specific file (e.g. `pmon/pmonroot.c`).
- *nvenv* : support code for a simple environment store in non-volatile RAM.
- *posix* : implements the generic POSIX "file i/o" interface functions, such as `open`, `close`, `read`, `write`, `ioctl`, `stat`, etc. They pass control to device-specific functions defined by the device files in the "memory file system" above.
- *paneldev* : implements the `/dev/panel` special device file, described in Section 19.1.5 "Alpha Display (/dev/panel)".
- *profil* : contains the profiling support functions which arrange to sample the program-counter at 100Hz.
- *sbrk* : is the rudimentary memory allocator required by `malloc()` et al. It dishes out consecutive, contiguous areas of memory between `_end` (the end of the program's data), and 64Kb below the stack. This hard-wired 64Kb stack size may be too small for some applications, and there is no check for the stack and memory pool colliding. You may need to change this limit!
- *signals* : is an emulation of the POSIX *signal* mechanism, which integrates with SDE's low-level exception handling.
- *timer* : is a generic interface to whatever timing hardware a board provides. It implements three high-precision interval-timers, modelled on the BSD / SVr4 `setitimer()` interface. It also maintains the current "elapsed" time for use by `time()` and `clock()`. One of the interval-timers is also used by the pc-sampling profiler.

- *tty* : handles i/o to ‘‘tty’’ devices (i.e. the console), including simple line editing, baud rate setting, etc. It implements a large subset of the POSIX *termios* interface.

21.1.1 Run-time Initialization

The startup code in `.../sde/kit/share/crt0.sx` sets up the initial run-time environment required by C and C++ programs. Its entry-point is `__start`, which is arrived at either by a jump from the end of the standalone `romlow` code, by an eval board's PROM monitor after your code has ben downloaded to RAM, or by *gdb* via an EJTAG probe or simulator. It performs the following steps:

- Initialises the *gp* register, required for *gp-relative* addressing.
- Moves the *sp* register to the the same address space (i.e. cached KSEG0 or uncached KSEG1) as the program's data has been linked for.
- Zeroes the ‘‘uninitialised’’ data section (*bss*).
- Initialises the POSIX i/o system and drivers, described above.
- Initialises the remote debug stub, if the **RDBG** symbol is non-zero. This may cause an immediate breakpoint if **RDBG** is greater than 1 (which is what happens if **`RDEBUG=immed`** is used in the example makefiles).
- Initialises the floating point coprocessor and/or CP1 emulator, as selected by the **#float** assertion (which is controlled by the the **`FLOAT`** variable in the example makefiles).
- Starts the profiling timer if **CFLAGS** contains the **-p** flag.
- Runs the C++ global constructors, if any. It uses `atexit()` to arrange for the C++ global destructors to be called when the program exits.
- Calls `main()`.
- If `main()` returns, then it calls `exit()` with the returned value as its argument.

21.1.2 Run-time Termination

The `crt0.sx` file also contains the low-level `_exit()` function, which performs the following steps:

- Calls the even lower-level `__exit()` function, defined in the monitor-specific directory. This will normally return control to the PROM monitor or *gdb*, or in a rommable program might switch off the board, or enter a tight loop.

21.2 Target-specific Code

Each target evaluation board or simulator has its own subdirectory under `.../sde/kit`. The list of supported targets is in Chapter 8 ‘‘Target Specific Libraries’’, and some historical and now unsupported targets are listed in Appendix E ‘‘Unsupported Targets’’. Each target's directory contains a configuration file `sbd.mk` which describes the key features of the targt, such as the CPU type, whether it has an FPU, the monitor type, the default download, ROM and RAM addresses, etc, etc. It also lists the files within that directory which handle board reset/initialization and devices (e.g. UART, timer, etc).

If you only want to run programs under control of an eval board's PROM monitor, then the board initialization code and UART driver can be omitted, since these functions are provided to your application by the monitor. If you do need to retarget SDE to a new board, then see Chapter 22 ‘‘Retargetting the Toolkit’’ for more details.

21.2.1 PCI Bus Configuration

The directory `.../sde/kit/pci/` contains generic PCI bus configuration, enumeration and access routines, which are included into the run-time system if `sbd.mk` defines **`PCI=yes`**. The functions in this directory then make use of board-specific functions to access the PCI bus controller chip; see `.../sde/kit/p6032/pci_machdep.c` for an example.

21.3 Monitor-specific Glue

Wherever possible the run-time system uses the low-level i/o facilities provided by a board's PROM monitor. It does this to:

1) Make it easier to retarget SDE to a new board which has a supported monitor.

2) Integrate more closely with the debugging facilities of the PROM monitor, so that you can use its interactive and/or remote debug facilities.

3) Make use of any remote console and file i/o facilities which it offers, while maintaining the standard POSIX and ISO / ANSI C "stdio" interfaces.

Like the board-support code, each supported monitor has its own sub-directory containing a configuration file `monitor.mk`, together with the monitor interface code. The directories are as follows:

Table 21-1 Supported PROM monitors

Directory	*Description*
`bare`	A "bare-board" interface for rommable programs, or for boards without one of the supported monitors. In this case software from SDE takes over the board devices and exceptions completely.
`yamon`	Interface to the YAMON monitor used on MIPS Technologies' development boards.
`mdimon`	Provides facilities (including virtual console and host file I/O) for programs running on targets connected to *gdb* via the "MDI" interface.
`mtspmon`	Provides facilities for SDE programs running on the Signal Processor side of a multi-threaded CPU, communicating with a Linux device driver on the Application Processor.
`gnusim`	Provides host file i/o for programs running on the GNU simulator included with SDE.
`idtsim`	Interface to the IDT/sim monitor used on boards supplied by IDT Inc.
`pmon`	Interface to the public-domain LSI PMON monitor, used on boards supplied by LSI Logic Inc. and other vendors.

21.4 Low-level CPU Management

The following files provide the low-level CPU initialization and control functions. In the supported, paid-for SDE version you'll find their source code in `.../sde/kit/share`; other users will find that the object code is supplied ready-built in the `.../sde/kit/free` directory, in a library file called *`SBD`*`.lib`.

- *cache.sx cache_ops.sx* : Interface layer to cache management functions, which can select at run-time between different cache architectures.
- *cp1emu.c* : A coprocessor 1 (floating point) instruction emulator, used when the coprocessor hardware is absent, or to handle those instructions which the coprocessor cannot (denormalised numbers, underflow, etc).
- *bremu.c* : is also required; it emulates branch instructions, which is a necessary part of emulating an FP instruction if they happen to be in a branch delay slot.
- *cw01cache.sx cw01cache_ops.sx* : Vendor-specific cache handling for the LSI CW400x/TR411x CPUs.
- *cw10cache.sx cw10cache_ops.sx* : Vendor-specific cache handling for the LSI CW401x CPUs.
- *cw10tlb.sx cw10tlb_ops.sx* : TLB initialization and management functions for the optional LSI CW401x memory management unit.

- *dbg.c* : The remote debug stub, used when debugging standalone, rommable programs, or when a board's PROM monitor does not implement the ''MIPS remote'' debugging protocol. See Section 21.4.3 ''Remote Debug Stub'' below.
- *dbgsig.c* : Dummy h/w interrupt initialization for remote debug stub; this can be overridden.
- *dbgsup.c* : Default i/o support routines for remote debug stub.
- *ecchandler.c* : Example cache/ecc error handler for R4000 SC/MC processors.
- *fcache.c* : Generic Flash ROM interface for the remote debug stub, allowing breakpoints to be set in Flash.
- *intrupt.c* : Generic, prioritisable interrupt dispatcher.
- *lr30cache.sx lr30cache_ops.sx* : Cache initialization and management for the LSI LR330x0 families.
- *m32cache.sx m32cache_ops.sx* : Cache support for the MIPS32 and MIPS64 architectures.
- *m32c1.sx* : Coprocessor 0 support for the MIPS32 and MIPS64 architectures.
- *m32tlb.sx m32tlb_ops.sx* : TLB initialization and management functions for the MIPS32 and MIPS64 architectures.
- *micromon.sx* : An ultra low-level, RAM-less ROM monitor program, which can be very useful when bringing up a new MIPS-Based design.
- *mipscp0.sx* : Low-level access to the coprocessor 0 registers, provided mainly for MIPS16 code which cannot use inline *asm* statements to access these registers.
- *muldivem.c* : A software multiply and divide instruction emulator for CPU cores that don't have the hardware multiplier unit.
- *noc1.sx* : Dummy floating point coprocessor functions for CPUs without an FPU.
- *notlb.sx notlb_ops.sx* : Dummy TLB functions for CPUs without a TLB.
- *r3kcache.sx, r3kcache_ops.sx, r4kcache.sx, r4kcache_ops.sx,* : r5kcache.sx, r5kcache_ops.sx" Cache initialization and management functions for the generic R3000, R4000 and R5000 families.
- *r54cache.sx r54cache_ops.sx* : Vendor-specific cache handling for the NEC R54xx family.
- *rc32cache.sx rc32cache_ops.sx* : Vendor-specific cache handling for the IDT RC32364.
- *rm7kcache.sx rm7kcache_ops.sx* : Vendor-specific cache handling for the PMC−Sierra RM7000.
- *r3kc1.sx, r4kc1.sx, r5kc1.sx* : Floating point coprocessor (CP1) initialization, register save/restore and control functions for the R3000, R4000 and R5000 families.
- *r3ktlb.sx, r3ktlb_ops.sx, r4ktlb.sx, r4ktlb_ops.sx* : TLB initialization and management functions for R3000-class and R4000-class memory management hardware.
- *romlow.sx* : The ''from reset'' initialization code, and boot exception handler. With the co-operation of board-specific functions this gets a rommable program to the point where the normal C run-time environment can be started. See Section 21.4.1 ''CPU Reset Handling'' below.
- *unaligned.c* : Unaligned-access exception handler and emulator.
- *watch.c* : Generic API to the CPU hardware watchpoint facilities, if available.
- *watchsup.c* : Support code for CPU hardware watchpoint facilities.
- *xcptlowb.sx* : Low-level MIPS exception handler.
- *xcptlow.sx* : Alternative low-level exception handler, for more complex environments.
- *xcptcache.s* : Example low-level R4000 ''cache error'' exception handler (see also *ecchandler.c*).
- *xcpt.c* : Higher-level exception support code, including default exception handler.
- *xcptshow.c* : Functions to report an exception status on the console.
- *xcptshowmin.c* : Functions to report an exception status on the console, small version.

21.4.1 CPU Reset Handling

The source file `.../sde/kit/share/romlow.sx` is used only when building a standalone, rommable program, and is compiled into a board-specific object file. Unsupported users get it in a pre-compiled object file in the board directories.

It is always linked at the beginning of the ROM, and this should be the virtual address where the CPU starts execution on a hardware reset – that is `0xbfc00000`, or `0xffffffffbfc00000` on a 64-bit processor, which map to physical address `0x1fc00000`.

It includes the following:

- A template showing one way to provide a moitor entry point table, should such a thing be required.
- The assembler code required to get a MIPS architecture CPU from a reset exception to the point of initialising the C/C++ run-time environment. Part of this is target-dependent, and is accomplished by calling the board-dependent `_sbd_reset` function, which is defined in the target-specific directory.
- The code to copy the instruction and read-only data segment from ROM to RAM. This copy is done only if the `.text` section has *not* been linked to start at the base of the ROM, and that is usually done only if you want to be able to set breakpoints in, and single-step through standalone programs. See Section 14.4.2 "Serial Debugging with SDE Debug Stub".
- The code to copy the initialised, writeable data section from ROM to RAM. The *sde−conv* program, when given the **−p** option, concatenates the initialised data segment to the end of the instruction and read-only data segment. See Chapter 17 "Manual Downloading".
- The code to re-vector *Boot Exception Vector* (BEV) exceptions to the address held in kernel reserved register *k0* (*$26*). Boot exceptions are used before RAM and caches have been tested and enabled (in normal operation the CPU vectors via cached RAM space, i.e. a low KSEG0 address). If *k0 == zero*, then it attempts to display a "Catastrophic Exception" message on the system console, indicating the location and cause of the error.

The file `.../sde/kit/share/ramlow.sx` is simply a dummy version of the `romlow.sx` file, which is used when building programs to be downloaded to RAM on a target with an existing monitor.

21.4.2 Exception Handlers

The files `.../sde/kit/share/xcptlowb.sx` and `xcptlow.sx` implement two alternative forms of the lowest level of exception handling for MIPS processors. Their job is to save the current processor state in a stack frame known as an *xcptcontext* (defined by *<mips/xcpt.h>*), set up a fresh run-time environment, and then call a C function. When the C function returns they restore the saved processor state and return to the interrupted program. Note that these low-level handlers neither save nor restore the floating point registers: your exception handling routines must explicitly call *fpa_save()* and *fpa_restore()* if they need to use, examine or modify any floating point registers. We recommend that exception level code should not perform floating point arithmetic!

The simplest and fastest handler is the default `xcptlowb.sx`. This handler remains on the current "application" stack, pushes a new *xcptcontext* frame, and then calls a standard C handler which does further dispatching to individual exception handlers (see `xcpt.c`, described below).

More complex run-time environments may need to use the `xcptlow.sx` handler, or some hand-crafted combination of the two. The `xcptlow.sx` file implements a separate exception-level stack, which is necessary if the stack pointer might not be valid on an exception (e.g. it may point to an unmapped address in KUSEG or KSEG2). Additionally the code uses a low-level dispatch table (`xcpt_astab`) which could allow certain exceptions to be handled quickly in assembler, without the overhead of saving/restoring a complete exception context (e.g. low-latency interrupt handling).

The higher-level exception handler is in file `.../sde/kit/share/xcpt.c`, and its associated header file is *<mips/xcpt.h>*.

21.4.3 Remote Debug Stub

When EJTAG is not available, remote debugging requires that the target board runs some sort of communications protocol which allows *sde−gdb* on the host development system to control and examine the program running on the target. This usually operates over a serial line, or perhaps over Ethernet.

When a program is being run under the control of a board's PROM monitor, and that monitor implements a supported remote debug protocol (which is true for the YAMON monitor, IDT/sim and LSI PMON), then you will probably use the PROM monitor's built-in remote debug support. See Chapter 14 "Debugging with GDB" for full instructions.

But if the program is running standalone (i.e. there's no separate monitor), or if your PROM monitor does not run a *gdb* debug protocol, then your program must have the remote debugging protocol code linked into it. This is implemented by the remote debug stub in `dbg.o`; if you have source code it will be in `.../sde/kit/share/dbg.c`.

If you use the example makefiles and their standard startup code then the debug stub will be automatically linked into your program, and initialised when both:

1) You are building a *rommable* version of the program, or the selected monitor does not implement a supported *gdb* remote debug protocol, and
2) The **RDEBUG** makefile variable is defined as "`yes`" or "`immed`". See Section 9.2 "Example Makefiles".

Once the debug stub has been initialised, it will then only take control if an unexpected CPU exception occurs. However if `RDEBUG=immed` was defined, then an immediate breakpoint is taken before your main program is started, to allow initial breakpoints to be set. See Section 14.4.2 "Serial Debugging with SDE Debug Stub" for more instructions on using *sde−gdb* with the remote debug stub.

Hardware-specific debug support

The remote debug stub contains some support for catching hardware interrupts, e.g. a debug button, or a Control-C (ASCII 0x03) received on the debug serial port. See the `_dbg_signals()` function in `.../sde/kit/P4000B/sbddbg.c` for an example of how to do this.

To support debugging of code in Flash memory, the debug stub performs all accesses to memory via a set of cover functions. See `_dbg_put_byte()` *et al* in `.../sde/kit/p4032/sbddbg.c`. You can also use the **−DSIMULATESSTEP** compile-time option to avoid having to rewrite a whole Flash sector on every single-step (see `.../sde/kit/share/dbg.c` for its effect).

It is also possible to integrate the debug stub with your own (perhaps interrupt driven) i/o system, by implementing your own version of the functions found in `.../sde/kit/share/dbgsup.c`

Multi-threading support

The remote debug stub does contain some support for debugging multiple threads/processes. See the dummy functions at the start of `.../sde/kit/share/dbg.c`. Contact us if you need to use this feature. These stubs can be overridden by a multi-threading kernel to provide thread debugging.

Retargetting the Toolkit

This section is a guide to retargetting or porting SDE to a new target board or simulator, and how to check your port with the example programs. While there's nothing to stop you doing this starting from SDE *lite* , one reason for supplying the run-time source code with the supported version of SDE is to help you to get your application up and running on a new MIPS-Based design with the minimum of extra programming. This section assumes you have all the files; unsupported users will have to figure things out for themselves.

Earlier in this document, Chapter 8 "Target Specific Libraries" listed the boards already supported by SDE. You should check with us before you do too much work; we might have already added the board that you want.

To add support for a new board you should:

1) Create a new directory in `.../sde/kit`, with the name of your board (e.g. "`MYBOARD`").

2) Copy into this directory all the files from the board directory `.../sde/kit/SKEL`.

3) Edit each of these files, as described by the detailed comments within them, to control your on-board devices.

In many cases you may be able to use existing, shared files for UARTS, timers etc, which are already used on other boards. There are many different boards and chipsets already supported: it is worth scanning other board support directories for sample code or simply for inspiration. In summary the files which you will need to create are as follows:

Table 22-1 Board-specific files

File	*Purpose*
`Makefile`	Trivial file which defines the board name and includes `../kit.mk`.
`sbd.mk`	Configuration file which describes the CPU type, endianness, presence of FPU, names of object files, memory map, etc.
`sbd.h`	Header file defining board-specific devices and registers, memory map, etc..
`sbdclock.c`	The low-level code to control the on-board timer. Most modern MIPS-Based CPUs (since the R4000 CPU) have an onchip counter and can use the common `r4kclock.c` driver; some other boards have drivers for offchip timers.
`sbdflashenv.c`	Support functions for storing board environment variables in Flash memory (if available).
`sbdfreq.c`	The low-level code to determine the CPU clock frequency. This is only strictly needed when using an on-chip timer, where `sbdclock.c` needs to know this value.
`sbdfrom.c`	Support code for Flash memory programming: recognises Flash memory address region and probes for Flash device.
`sbdfrom.h`	Defines the layout and type of Flash memory device(s).
`sbdmem.c`	Describes the physical RAM layout for memory allocation; only required if it is not contiguous.
`sbdmisc.sx`	Miscellaneous low-level functions like `mips_wbflush()`.
`sbdnvram.c`	Support functions for storing board environment variables in non-volatile memory (if any).
`sbdpanel.c`	Low-level code to display simple messages on on-board or front-panel LED alphanumeric display.
`sbdpci.c`	Support functions for initialization of host PCI bus controller (if any) and configuration of PCI devices.
`sbdreset.sx`	The code to initialise the on-board memory controller and any other board-specific reset code. This is only necessary if you intend to build standalone (i.e. rommable) programs.

File	*Purpose*
`sbdser.sx`	A simple driver for the board's UART. Again this is usually only necessary for standalone programs; other programs will use the PROM monitor's i/o routines.
`sbdtime.c`	For boards that have a battery-backed real-time clock this file computes the current time in seconds since 00:00:00 Jan 1, 1970.

Fortunately, if you already have a supported PROM monitor running on the board (e.g. the YAMON monitor, PMON or IDT/sim), or are running on a supported simulator, then many of these files can be dummied out; the monitor/simulator handles the power-on initialization and console i/o for you. The only board-specific files that require real code are `sbdclock.c`, and possibly `sbdfreq.c`, which are required to implement the interval timing functions (which you will need for benchmarking and profiling).

When performing a full port, then in order to support rommable code, particular care must be taken in the `sbdreset.sx` and `sbdser.sx` files. Until the generic code in `.../sde/kit/share/romlow.sx` has completed its job, then memory may not be used to store variables or a stack (it may not be enabled yet, and/or may have to be cleared to initialise parity, etc). The caches and FPU will also not be initialised yet, and cannot be used. The board-reset and low-level serial i/o code must therefore be capable of operating only in registers. Also tricky is that these functions (and anything which they call) must be position-independent because, until they are relocated, they may not at first be running at their final link address: absolute jumps may not be used, only branches and *bal* for subroutine calls. If you have to load the address of a code label or read-only data label, then you must add register *s8* which holds the relocation factor, e.g.

```
la      a0,reset_tab
addu    a0,s8
lw      t0,0(a0)
```

Having created the new files and got them to compile, you can test them with some of the example programs:

- *Micromon* : built automatically as part of the board-support kit, it can be used test the reset and serial i/o code even before a new board's memory controller is working. The ultra low-level monitor interprets a "reverse polish" stack-based command language allowing you to probe devices and memory – press '`?`' for help.
- *Kittest* : should be used to check that the low-level serial i/o code as part of the full C environment.
- *Minimon* : the mini command-line monitor has a number of builtin commands which can be used to check out many of the remaining functions, as follows:

  `cache`: should report the correct cache sizes.

  `stat`: should display the correct memory size and CPU frequency.

  `time`: should display the correct date and time, if you have a real-time clock chip.

  `itimer`: checks that the timer support code is returning monotonically increasing values, and interrupting at the correct rate; it should run for exactly 120 seconds (check it with a stopwatch).

  `ls /dev`:
  Directory listing should include "`flash0`" etc. if you have implemented Flash memory support; and "`panel`" if you have implemented front-panel display support.

  `echo wow! /dev/panel`:
  Should display "wow!" on your front-panel display, if implemented.

  `dump /dev/flash0 0 16`:
  Should dump the first 16 bytes of your Flash memory, if detected.

- *Flash* : the Flash memory test/example should report each of your Flash memory devices, and run through to completion without any errors, if you have implemented Flash memory support correctly.
- *PCI* : the PCI test/example should enumerate and list all devices on your PCI bus, if you have implemented the PCI support code correctly.

22.1 Common Device Files

There are a number of files in `.../sde/kit/share` which provide support for common UART and timer chips. You may be able to use these directly for your board, by *#include*-ing them into your files, or simply use them for inspiration:

- *m82510.s* : driver for the Intel M82510 serial controller.
- *mk48t02.c* : support for the Mostek MK48T02 clock/calendar.
- *mpsc.s* : driver for the NEC uPD72001 serial controller.
- *ns16550.s* : driver for the NS16450/16550 UART.
- *r361clk.c* : interval timing support for the IDT R36100 on-chip timer.
- *r4kclock.c* : interval timing support for the on-chip timer found on most modern MIPS-Based CPUs; relies on the on-chip timer interrupt being enabled by your hardware engineer.
- *s2681.s* : driver for the Signetics SCN2681, Motorola 68681 and UMC UM26811 DUART.
- *s2681clk.c* : interval timing support using the timer on the S2681 DUARTs.
- *vacser.s* : driver for the serial-port on the VAC068 VME-bus controller.
- *z8530.s* : driver for the Z8530 DUART.

Known Problems / Errata

Known problems with the tools are listed below.

Compiler

- When compiling with **–mips16** an initialised module level static variable which is then redeclared at function level with an extern will generate a linker error. For example the following would fail:

```
static int answer = 42;

int question ()
{
    extern int answer;
    return answer;
}
```

- The SDE v5 **–mno–gpopt** compiler option is no longer available. This means that you must compile 32-bit modules with **–G0**, if they might be linked with code compiled for the MIPS16 ASE.

Debugger

- Debugging of C++ programs is impeded when the linker's **––gc–sections** option is used.
- Remote serial debugging of MIPS16 code is not possible. Other debugging targets such as MDI (i.e. the MIPSsim simulator and FS2 EJTAG probe), and GNU simulator, are not affected.
- The MIPSsim simulator does not yet support the new *team* mechanism for synchronized debugging using separate copies of GDB.
- You will need version 2.1.8.0 (or higher) of the FS2 probe software to use the new *team* and *group* mechanisms with a 34K core.

Example Programs / Kit

- The `flash` example hangs when run on the Malta platform. This will be fixed in a future release.

GNU simulator

Caches, write buffers, exceptions, timers and other i/o devices are not emulated – only a raw CPU, FPU, PROM monitor and RAM are emulated. It emulates a large range of MIPS architecture processors, but stops short of emulating exceptions, so it isn't suitable for OS development. The MT ASE is also not supported. If you're looking for a free MIPS simulator capable of supporting OS development consider QEMU.

Download tools

The *sde–conv* program produces a range of output formats for various PROM monitors and EPROM programmers, but it may not include the particular format that you need. The source code is supplied in the `convert` directory of the source tarball, and it is easy to add new output formats if required.

Getting Support

MIPS® Software Toolkit customers have a direct line to the MIPS support desk. SDE *lite* users are not entitled to support but, while we don't offer a guaranteed response, may send questions to software@mips.com.

To help us to help you, try to do the following:

- So that we know who you are and what software you have, please quote your support account ID, if you have one.
- If you have a program which you believe is building incorrectly, do what you can to reduce the size of the example which shows the problem. Then where possible send us:
  a) Details of host operating system on which you are running the tools, and your current environment. On UNIX hosts the output from **uname –a** and **env**; on Cygwin use **cygcheck –srv**.
  b) The version of the tools which you are using, this can be obtained by running **sde-gcc –v**.
  c) The complete command line that triggers the bug.
  d) Any warnings or error messages from the tools.
  e) In the case of a compiler problem, the preprocessed file ("*.*i**") that triggers the bug, generated by adding **–save–temps** to the complete compilation command – this allows us to reproduce your problem without having to completely duplicate your build environment.
  f) In the case of a problem with the binary utilities or linker, then the set of object files and libraries which trigger the problem, as a compressed tarball or zip file.
  g) If you think you've found a problem with the run-time libraries, then a small example which can be run on a simulated target (e.g. the GNU or MIPSsim simulators).

If you do not have access to Internet, then we can be contacted by fax at (+1) 650 567 5150.

Upgrading

Any SDE *lite* user user can upgrade to the MIPS® Software Toolkit at any time; you'll get MIPS Technologies' support and updates, and more source code. You should think seriously about doing this if you've used SDE *lite* for evaluation and are moving on to product development.

To upgrade just contact us at tool.sales@mips.com.

Internet data at MIPS Technologies

We are accessible on the World Wide Web; here you can find documentation, upgrade information, and much more. Visit http://www.mips.com, and follow links to "Products" and "Software Tools"

Related Services

You may be interested in MIPS Technologies' training services, details of which can be found on our website http://www.mips.com, or by sending us an email to training@mips.com.

References

[Sweet99]
See MIPS Run, Dominic Sweetman (of MIPS Technologies), 1999, Morgan Kaufman, ISBN 1-55860-410-3.

We have to give special mention to this comprehensive guide to the MIPS Architecture and programming; firstly because one of us wrote it, and secondly because if you read it carefully enough we'll save time on support work.

[Farq94] *The MIPS Programmers Handbook*, Farquhar & Bunce, 1994, Morgan Kaufmann, ISBN 1-55860-297-6.

Example-based programming book aimed at small MIPS-Based systems.

[SGI96] *MIPSpro™ Assembly Language Programmer's Guide*, Silicon Graphics Inc.
http://techpubs.sgi.com/library/tpl/cgi-bin/browse.cgi?db=bks&cmd=toc&pth=/SGI_Developer/MProAsLg_PG

[Kane92] *MIPS RISC Architecture*, Gerry Kane and Joe Heinrich, 1992, Prentice Hall, ISBN 0-13-584210-7.

Reference manual to MIPS instructions, focussed on the machine instruction level.

[Kern88] *The C Programming Language (Second Edition)*, Brian W. Kernighan and Dennis M. Ritchie, 1988, Prentice Hall, ISBN 0-13-110362-8.

Throw away all those cheerfully coloured fat books with big letters and lots of pictures, if you want to program in C you need this and nothing else.

[Lewine91]
POSIX Programmer's Guide, Donald Lewine, 1991, O'Reilly, ISBN 0-937175-73-0

An introduction to and complete set of manual pages for the POSIX.1 programming interface, of which the SDE run-time system implements a generous subset.

Then there are reference works; we need to put these in, but you won't read them unless you have to:

[POSIX88]
IEEE Standard 1003.1-1988, Institute of Electrical and Electronics Engineers Inc., 1985.

[ABI] *System V Applications Binary Interface − Revised Edition*, Unix System Laboratories, Prentice Hall, ISBN 0-13-877598-2.

[MIPSABI]
System V ABI MIPS Processor Supplement, Unix System Laboratories, Prentice Hall, ISBN 0-13-880170-3.

[ELF] *Understanding ELF Object Files and Debugging Tools*, Mary Lou Nohr (Editor), Prentice Hall, ISBN 0-13-091109-7

[MD00410]
MIPS® SDE for Linux Getting Started Guide, MIPS Technologies, Inc.

The document which describes the SDE toolchain configured for native development Linux/MIPS kernels and applications.

[MD00374]
MIPS32® Architecture for Programmers Volume IV-e: The MIPS® DSP Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

[MD00378]
MIPS32® Architecture for Programmers Volume IV-f: The MIPS® MT Application-Specific Extension to the MIPS32® Architecture, MIPS Technologies, Inc.

You can't (so far as we know) buy the following GNU manuals, but they're provided as part of SDE.

[Binutils] all the object-code tools except the linker itself, which gets a separate manual [Ld].

[Conv] the SDE-specific ELF file conversion tool (*sde−conv*).

[Cpp] the GNU C pre-processor; only for specialists.

[Gcc] the compiler manual. Serious users should think about reading this through one time.

[Gdb] the debugger. Probably for reference only.

[Gprof] the profiler; read this if you're planning to do performance analysis.

[Ld] the linker; read this if you need to go beyond the tricks used in SDE examples.

[Make] read this if you're keen to create makefiles even more exciting than those in the examples.

[Stabs] documentation on the data structures used to pass debugging information.

Appendix A: Copyrights

Many of the utilities contained in this package are derived from Free Software Foundation code, whose *GNU General Public Licence* obliges us to make their source code and our changes to it available to anyone that requires it. Please read the file `.../COPYING` for more details on FSF terms and conditions. All the GNU sources, including our enhancements, are published on our public FTP server.

The only GNU libraries included with SDE are `libstdc++` and `libgcc`. The `libstdc++` library copyright includes this special proviso:

```
As a special exception, you may use this file as part of a free software
library without restriction.  Specifically, if other files instantiate
templates or use macros or inline functions from this file, or you compile
this file and link it with other files to produce an executable, this
file does not by itself cause the resulting executable to be covered by
the GNU General Public License.  This exception does not however
invalidate any other reasons why the executable file might be covered by
the GNU General Public License.
```

The `libgcc` library includes this special proviso:

```
In addition to the permissions in the GNU General Public License, the
Free Software Foundation gives you unlimited permission to link the
compiled version of this file into combinations with other programs,
and to distribute those combinations without any restriction coming
from the use of this file.  (The General Public License restrictions
do apply in other respects; for example, they cover modification of
the file, and distribution when not linked into a combine executable.)
```

The GNU license terms do not apply to the SDE “kit” run-time system, C library, maths library and IEEE–754 emulation library, which are Copyright (c) MIPS Technologies, Inc. All rights reserved. Where appropriate see the copyright headers in the individual source files.

Software developed using SDE is free of any code which would make it subject to GNU license conditions (i.e. GPL or LGPL).

Appendix B: MIPS™ Freedom-to-Use License

For full details of your rights and obligations regarding the use of derived binaries see Version 2 of the MIPS "Freedom-To-Use" license agreement in file `.../MIPS-FTU`.

Appendix C: Release History

See Chapter 4 "Information for Upgraders" for a description of the important user-visible changes in SDE v6.

Release 6.06.00 Update

- Adds support for the new 74K core family. The **–mtune=74k***[cfx]* option schedules code for the 74K pipeline; the **74kx** variant option schedules for 1:1 FPU, otherwise a 1:2 FPU is assumed.
- The compiler has new builtin intrinsics to support revision 2 of the DSP ASE. These new intrinsics are enabled automatically when you specify **–march=74k***[cfx]*.
- The `strcmp()` and `memcpy()` C library functions have been tuned to further improve their performance on current MIPS cores.
- The compiler now defaults to **–mno–embedded–data**. The effect of this is that constant data that used to be placed in the `.rodata` section may be moved to `.sdata`, where it can be accessed more efficiently using gp-relative addressing. The MIPS16 **–mno–data–in–code** option also requires **–mno–embedded–data**.
- The decompressing loader example (*zload*) makefile now builds a tiny application and compresses it; the application will then attempt to load and execute it.
- Improved GDB multi-VPE debugging: fixed problem with the initial reset of the target when multiple VPEs are connected.
- The SDE libraries use a new API to implement thread-safety and reentrancy, in place of the Pthreads API used previously. Any applications which have been built with the SDE C/C++ libraries, but without the standard SDE "kit" library, may now need to replace their *Pthread* stub code with the new *sdethread* interface described in `.../sde/include/sdethread.h`. An example single-threaded implementation is included in pathname sde/kit/share/stubs.c .
- The SDE build system now supports the integration of separate additional Thread Support Packages (TSP), providing a fully thread-safe run-time environment that can be linked with a third-party real-time microkernel. The first TSP offering is for Express Logic's ThreadX® RTOS – please contact your MIPS Technologies representative for more information and to order.
- There are new board targets including `MALTA32R2*` for MIPS32 Release 2 targets, `MALTA32MT*` for MIPS MT ASE, and equivalents for `MSIM`. For consistency the `MALTA32F64` targets are replaced by the new `MALTA32R2` targets with an 'F' suffix. Additional 'J' variants (providing virtual host i/o via EJTAG) to cover all Malta targets. See Chapter 8 "Target Specific Libraries".
- The SDE Makefile system will now warn you if you request a "FEATURE" which is not implemented by your chosen board support package. A requested feature can be marked as optional by prepending a '/' character.
- HP–UX is no longer a supported host.

Release 6.05.00 Update

- The MIPS16 **–mno–data–in–code** option is now supported by a new multilib library variant. The use of **–mno–data–in–code** may be needed if running code in split onchip I & D SRAM/SPRAM, except on the M4K, or 4KEc with external D to I redirect, in which case **–mcode–xonly** should be used instead. See Section 12.7 "MIPS16® ASE support".
- The MIPS16 libraries are now built with **–mcode–only** as the default.
- Added support for unordered floating point comparisons in MIPS16.
- GDB now supports the concept of a device *group*, which allows debugging of a single program image running on multiple VPEs or cores in parallel in a unified memory system, e.g. an SMP operating system. The associated *team* mechanism supports synchronized debugging by multiple legacy debuggers of independent applications or operating systems running on separate VPEs or cores.
- The board support kit now includes new configurations for size-optimized MIPS16 builds which use hardware floating point, namely `MSIM16FB`, `MSIM16FL`, `MALTA16FB` and `MALTA16FL`.

- The board support kit now includes and uses its own linker scripts instead of relying on the linker-provided scripts.
- The old `24kfx` and `34kfx` CPU names are deprecated in favor of `24kx` and `34kx`, for compatibility with upstream GCC 4.
- SDE is now distributed under the terms of the updated Version 2 of the MIPS Freedom-To-Use license. See the file `.../MIPS-FTU`.

Release 6.04.00 Update

- **Incompatible Change**: The assembler and the disassembler have swapped the `MFTR` instruction's register operands, to adhere to the latest MT ASE specification. This affects only the raw `MFTR` instruction; the `MFTLO`, `MFTHI`, `MFTACX`, `MFTDSP`, `MFTC1`, `MFTHC1` and `CFTC1` "idioms" remain unchanged. The `MTTR` instruction is also unaffected.
- **Incompatible Change**: For MIPS32 the compiler now uses registers `$f0` and `$f2` to return *complex float* values from functions, instead of `$f0` and `$f1` which was incompatiable with the original *O32* ABI.
- The DSP control register's SCOUNT and POS fields are now treated as global registers, and may therefore be safely used after a function call which sets them, or *vice versa*. Other DSP control register fields remain local to a function, which means that their values are undefined after a function call. Previously all DSP control register fields were treated as local.
- Improved scheduling by the compiler of MIPS DSP ASE intrinsics.
- The compiler's **–frename–registers** option is now enabled automatically at **–O2** and **–Os** (previously only at **–O3**). This results in better register allocation, particularly for small register classes such as the DSP accumulators.
- The DSP accumulator registers are now represented in the Dwarf debug data, so variables which are assigned to DSP accumulators can now be accessed via GDB.
- The compiler can now make use of the MIPS32 `MSUB` and `MSUBU` instructions.
- The **–ffunction–sections** option can now be used when compiling with **–mips16**.
- Improved handling of mixed 32-bit and MIPS16 code which uses hardware floating point.
- Several fixes to GDB, mainly associated with using MIPS16 with hardware floating point.
- Multi-VPE debugging of the 34K CPU family is now supported by GDB in conjunction with recent versions of MIPSsim and the FS2 EJTAG probe. You can run two copies of GDB or Insight, each connected to a separate VPE as if they were independent CPU cores. See Section 14.2.2 "Debugging Multiple VPEs".
- The new '`mdi threadstepall`' setting replaces the previously inconsistent use of '`mdi stepinto`' to enable single-stepping of all TCs in parallel.
- The assembler now handles branches between different object code sections defined within the same source module.
- The GNU simulator now supports the MIPS DSP ASE instruction set.
- The SDE compiler, libraries and header files have now been validated with the C90 (ISO 9899:1990[1992]) subset of the "Plum Hall Validation Suite for C". This process resulted in several fixes to the libraries and header files.
- The C library and the "kit" board support libraries have been restructured to reduce significantly the ROM footprint of production code. See Section 11.1.3 "Minimal C library", and the comments in the *hello* example's Makefile for more information on the available build options.
- The "kit" now includes new board support configurations for size-optimized M4K builds, namely `MSIMM4KB`, `MSIMM4KL`, `MALTAM4KB` and `MALTAM4KL`. These configurations remove support for features that are not present in the M4K, such as caches, TLB and FPU. They also remove the sign-on messages and early-life exception handling in the ROM startup code, and select the MIPS16 instruction set.

- There are new board configurations which build MIPS16 versions of the board support libraries, but are otherwise fully-fledged configurations, namely `MSIM16B` and `MSIM16L` for devices without an FPU, and `MSIM16FB` and `MSIM16FL` for those with.
- The MIPS16 C libraries now have versions of the `memcmp`(), `memcpy`(), `memset`(), `strcmp`(), `strcpy`(), and `strlen`() functions designed for a balance of size and speed, rather than the highest possible performance.
- The SDE libraries are now compiled with the **–ffunction–sections** flag, and the example programs linked with the **–gc–sections** option, to automatically delete unused functions from the application and libraries.
- The POSIX-compatible i/o system is no longer automatically linked into an application which only performs i/o using the *stdin*, *stdout*, and *stderr* streams. In such cases the *stdio* functions will use the low-level console i/o facilities of the board ‘‘monitor’’. One side effect of this is the loss of POSIX-style input processing (backspace, control-c, etc) when reading from the console. Any call to `open`(), `fopen`() or `fdopen`() will cause the POSIX-compatible i/o system to be included.
- The `mdimon`, `mtspmon` and `gnusim` POSIX ‘‘file systems’’ now include a `/tmp` directory, which provides direct access to the `/tmp` directory on the host computer, permitting use of the standard `tmpfile`() and `tmpnam`() library functions.
- There is a new `_sbrk`() function which allocates heap memory, but doesn’t zero the allocated region as required by `sbrk`(). The new function is used by `malloc`() et al, which are not specified to return zeroed memory, speeding up the allocation of large buffers.
- Exception and interrupt handling now work reliably when running under YAMON.
- The example Makefiles no longer support the `CRT0FLAGS` variable. The `crt0` C run-time startup module is now largely controlled by link-time symbol definitions. The profiling ‘‘`mcrt0.o`’’ version of the C run-time startup object file is no longer used – profiler initialization is performed automatically when any linked module was compiled with **–p** or **–pg**.
- The example Makefiles no longer support the `TIMING` variable.
- The example Makefiles now generate a linker map file automatically, named after the executable file with ‘‘`.map`’’ appended.
- The example Makefiles now record the last-used value of **SBD**, and will reuse that if a new value is not specified on the next use of **`sde-make`** in the same directory.
- It is no longer necessary to run **`sde-make depend`** to generate header file dependencies. The example Makefile system now creates and maintains dependencies automatically, so long as you specify the `SRCS` variable in your Makefile. The `OBJS` variable is now optional, and defaults to being derived automatically from `SRCS`.
- The ‘`sdesetup`’ script, and the ‘`mdi`’ command and its associated configuration tools now support line editing when run by a Bash shell.

Release 6.03.01 Update

- Various bug fixes to GDB.
- The linker now regards non-matching HI16/LO16 relocations as being safe and does not generate an error.
- Fixes a potential alignment problem when optimizing using MIPS16 section-relative addressing. The compiler now adjusts the section relative base to be aligned to the largest alignment found within the section.

Release 6.03.00 Update

This release is intended to support GA of the 34K and 24KE cores.

- The compiler now schedules MIPS DSP ASE instructions – as generated by the builtin DSP intrinsics – more accurately for the 24KE and 34K pipelines. Similarly for the conventional MIPS32 multiply and multiply-add instructions.
- The compiler generates smaller and faster code for MIPS16 functions with large stack frames. Accessing correctly aligned byte/halfword fields in small automatic struct variables generates better code. The MIPS32

Release 2 bit insert, extract and rotate instructions will now be used automatically in more cases, and the `nmadd` and `nmsub` instructions are now available when compiling with the **–mips32r2 –mfp64** options.

- The calling convention for the **–mips32r2 –mfp64** combination now conforms more closely to the O32 ABI. Any code built by previous releases of SDE v6 using these options should be recompiled before linking with the libraries supplied in this release.
- Selecting the **–msmartmips** ASE option now works reliably, and the GNU simulator can now simulate SmartMIPS instructions.
- GDB is more reliable when debugging multi-threaded code on a 34K CPU or 34K MIPSsim. However multi-VPE debugging (multiple GDBs debugging a different application on each VPE) is not yet fully supported by any of the MDI targets – full support for this feature will be added in a future maintenance release.
- GDB now outputs the floating-point exception flags and condition codes in response to the `info float` command. To see the floating point data registers use `info all` or `info reg float`.
- GDB hardware watchpoints now work correctly and at full CPU speed in conjunction with an MDI-connected EJTAG probe.
- GDB now treats addresses in the unmapped kseg0 and kseg1 regions as overlaying each other, so Insight can continue to display source code correctly when execution switches between the two regions.
- Insight can now scroll the memory window as expected.
- Insight will now update the register window when a register is modified via the command-line. Coprocessor registers which are not available on a particular CPU will be displayed as ‘‘blank’’. Register values are now right aligned within their cells instead of centred.
- Printing a floating point register in GDB (e.g. `print $f0`) will now display it as a pseudo ‘‘union’’ of double-precision, single-precision, word and long integer, making it easier to interpret for different contexts.

Release 6.02.03 Update

- The ‘‘v’’ (variable) variants of the intrinsics for the DSP ASE have been removed. The compiler will automatically generate the immediate version of the instruction if the operand is a constant within the appropriate range – otherwise it will load the value into a register and use the variable version of the instruction.
- The DSP ASE’s `repl.ph` assembler instruction, and its matching compiler intrinsic, now accept a signed immediate in the range -512 to +511, instead of 0 to 1023. This is not a change to the DSP ASE, but the assembler and compiler syntax have been brought into line with the definition of the instruction.
- The `absq.ph`, `extl.w`, `extl_s.w`, `extlv.w` and `extlv_s.w` instructions have been deleted from the DSP ASE, and removed from the assembler, disassembler and compiler intrinsics.
- The intrinsics for the DSP ASE will be enabled automatically if the **–march=** compiler option is used to select one of the following CPU types: `24ke`, `24kec`, `24kef`, `34k`, `34kc`, or `34kf`. Otherwise **–mdsp** is still required.
- The **sde–objdump** disassembler can now display interleaved source code and instructions, when requested to do so by the **–S** or **––source** flags.

Release 6.02.02 Update

- GDB supports debugging of multiple hardware TCs in applications built using the the MIPS MT ASE, in conjunction with 34K MIPSsim. See Section 14.2.1 ‘‘Debugging LLMT Applications’’.
- GDB supports hardware data and execution breakpoints in conjunction with the MDI interface (i.e. with MIPSsim and EJTAG probes). This enables the use of GDB’s `rwatch`, `awatch`, and `hbreak` commands, and `watch` to be run at full speed rather than being emulated.
- The SDE AP/RP bare-iron kit now supports debugging of the RP program using SDE’s GDB remote debug stub. A new example which demonstrates the use of interrupts inside the RP box is provided. See Section 14.2.3 ‘‘Debugging AP/RP Applications’’.

- For consistency GCC now recognises **–mtune=24ke** as an alias for **–mtune=24kec**.
- The MIPS DSP ASE's `extp` and `extpdp` assembler instructions, and their matching compiler intrinsics (see Section 18.10 "Intrinsics for MIPS® DSP ASE") previously accepted an immediate value in the range 1 to 32 for the *size* operand, however they now accept values in the range 0 to 31 (i.e. *size*-1), so as to match the the `extpv` and `extpdpv` instructions.
- The *minimon* example program is now capable of loading, relocating and executing a position-independent ELF Dynamic Shared Object (DSO) file. The directory contains a simple example.
- The *sdesetup* and MDI *fragbuilder* scripts now recognise and operate correctly on AMD64 Linux hosts.

Release 6.02.01 Update

- Now supports the "`"mips16`" and "`nomips16`" function attributes, see Section 12.7 "MIPS16® ASE support".
- Recognises **4ksc** and **4ksd** cpu types for **–mtune** option, as aliases for **4kc** and **4kec** respectively.
- Additional MIPS16 code size optimizations.
- Supports *complex* floating point return types in MIPS16 "hard-float" code.
- The **–mips16** option is now ignored when compiling assembly language source files using the `sde-gcc` front end. If you really want to write MIPS16 assembly language, you must add "`.set mips16`" to your source file. This behaviour matches previous SDE v4 and v5.

Release 6.02.00 Update

- The toolchain now supports the MIPS16 ASE, including cores which also have a hardware FPU, such as the 24Kf.
- Adds support for the new 34K and 24Ke core families (**–mtune=34k** and **–mtune=24ke**).
- The assembler, disassembler and debugger are now compatible with MT ASE v0.971 and DSP ASE v0.98.
- The compiler includes a set of built-in "intrinsics" which allow C and C++ code to generate instructions from the DSP ASE. See Section 18.10 "Intrinsics for MIPS® DSP ASE".
- The kit includes run-time support code for applications built to run on the virtual Signal Processor of a multi-threaded CPU running the Linux operating system on its virtual Application Processor. The new board support kits are named `MALTA32LSP`, `MALTA32BSP`, `MSIM32LSP` and `MSIM32BSP` for the Malta and MIPSsim targets, little- and big-endian.
- The examples directory contains the new `rtlx` example program, which is intended only to demonstrate use of the Linux AP/RP interface (`mtspmon`), and will only work with the kit configurations listed above.
- Applications built for the MIPSsim kit configurations (`SBD=MSIM*`) will now be built in the "ram" format, for running directly in the simulated RAM, and without the "rom" CPU initialization code. This can be done because MIPSsim initializes the simulated caches, TLB, RAM, etc – which would be not be the case on a true hardware CPU.
- The ISO C99 <*fenv.h*> header file is now provided, together with the associated functions in the maths libary (**–lm**) which provide control over floating point rounding mode, status flags and exception handling.
- The new <*sgidefs.h*> provides symbols which define the currently selected MIPS ISA and ABI, in a form which is compatible with GCC, and the Irix and Linux operating systems.

Release 6.01.02 Update

- The compiler, assembler and debugger now support the 64-bit MIPS64 ISA, using the "N32" calling convention. See Section 12.6.1 "64-bit Calling Conventions".
- The board-support kits for MIPS64 targets are available again – see Table 8-1 "Supported target boards and simulators".

- The include file *<mips/asm.h>* now defines the assembler macros described in SGI's *N32 ABI Handbook* for creating stack frames in an ABI-independent way, e.g. `NESTED`, `REG_S`, `REG_L`, `ALSZ`, `ALMASK`, `PTR_ADDU`, etc.
- The SDE kit run-time library now generates all tables of machine exception numbers, names, and associated POSIX signals from the central CPU-specific header file (e.g. *<mips/m32xcpt.h>*), rather than scattered ifdefs which were difficult to coordinate. The bulk of the exception context stack frame is now identical for all CPUs, with CPU-specific registers stored at the end. Any CPU-specific exception save/restore code is added to the generic `xcptlowb.sx` handler using hooks defined in the CPU-specific header file. Any assembler source code which used the C preprocessor conditionals (e.g. "`#ifdef`") to detect the presence of a particular exception name (e.g. `XCPTCEU`), must now use the assembler's "`.ifdef`"; C code can no longer use preprocessor conditionals to test for specific exception names.
- The standard `ffs()` C library function (find first set bit) has been joined by `ffsl()` and `ffsll()`, which take a *long* and a *long long* argument respectively. With MIPS32 and above the compiler will inline these using the **`clz`** or **`dclz`** instruction .
- A bug in the C `strncat()` library routine has been fixed.

Release 6.01.01 Update

- The assembler, disassembler and debugger are now compatible with the MT ASE v0.97 and DSP ASE v0.97.
- A disassembler bug which caused certain floating point instructions (`madd`, `msub`, etc) with certain operands, to be displayed as coprocessor 3 instructions has been fixed.
- GDB now supports thirty two 64-bit floating point registers with the MIPS32 Release 2 ISA (i.e. programs built with **–mips32r2 –mfp64**).
- The GDB Insight GUI now works on Windows hosts.
- The flash memory library support code now works when compiled by GCC 3.4.
- Fixes a bug in the run-time profiling code which caused an address exception when the code segment was an odd multiple of 4 bytes.
- The example makefile system now links the co-dependent run-time libraries as a *group* (i.e. using the **––start–group** and **––end–group** options).

Release 6.01.00 Update

- Major new release based on an up-to-minute GNU toolchain, with extensions, tuning, integration and packaging by MIPS Technologies. These are the closest base releases:

gcc	3.4.2
binutils	2.15.92
gdb	6.2
make	3.80

- Adds assembler-level support for the DSP ASE. The new *<mips/dsp.h>* header file defines the DSP ASE registers and bit-fields.
- Adds assembler-level support for the MT ASE. The new *<mips/mt.h>* header file defines the MT ASE registers and bit-fields.
- Adds compiler support for the paired-single SIMD vector floating-point format, and the MIPS-3D ASE. See the GCC manual for more details.
- Libraries for the "legacy" ISAs MIPS I to MIPS IV are no longer supplied – only libraries for MIPS32 and onwards are included.
- As a temporary measure this release does **not** support the MIPS64 ISA. This will reappear in an upgrade release in the near future.

- As a temporary measure this release does **not** support the MIPS16 ASE. This will reappear in the 6.02 release.
- GDB MDI back-end:
  - Now supports a list of profiling region start/end labels, so as to support profiling of programs from foreign toolchains. See the "`set mdi ftext-symbols`" and "`set mdi etext-symbols`" commands.
  - The debugger can now be stopped (by pressing the Stop button, or Ctrl-C in the command line version), even when an attached program is calling MDI file i/o system calls continuously.
  - GDB no longer crashes if an MDI connected program exits with an exit value not in the range 0 to 255.
  - The auto-generated MIPSsim config file can now be made to reference a user-written device config file, using "`set mdi devcfgfile`".
  - The "`mdi set/show cache`" and "`mdi set/show cp0`" commands now accept a full GDB expression for the arguments, not just simple constants.
- Installation scripts:
  - More reliable in face of being installed in unexpected locations.
  - Handle MIPSsim installations in the Windows (non-Cygwin) file system better.
- The sde-conv tool no longer generates spurious NULL program header entry in .relf file, and sets PADDR field to same as VADDR field.
- Run-time libraries, headers and examples:
  - Major rewrite of software floating-point library giving significant boost to programs compiled with **–msoft–float**.
  - Kit and example Makefiles enhanced to work with both SDE 5 and SDE 6 toolchains. The `CPUVARIANT` variable is now just a CPU name, and not a compiler option.
  - You can now generate a simplified example Makefile, derived from the current SBD settings, which can then be easily modified for standalone applications. See Chapter 10 "Porting an ISO / ANSI C Program".
  - New header file <*mips/mips32.h*> defines application-level "asm macro" intrinsics for the MIPS32 and MIPS32 Release 2 instructions which are not supported by the compiler.
  - Similarly for <*mips/mips64.h*>
  - New header file <*mips/mips4ks.h*> defines the extra CPU-specific coprocessor registers provided by the MIPS Technologies 4KS family of cores.
  - The new `SDETOP` Makefile variable can be used consistently to point to the top of the SDE kit/examples/lib/include tree, which allows applications to be built outside of that tree; this supercedes the old DEPTH variable which was confusing.
  - The `PROFILE` Makefile variable can now be set to "`feedback-generate`" to build a profiled program, and then "`feedback-use`" to use the resulting profile data when rebuilding the application.
  - The Dhrystone application Makefile now explicitly enables loop unrolling, but disables function inlining, to conform to Dhrystone rules.
  - The CorExtend header file <*mips/udi.h*> gains the `mips_udi_rwi()`, `mips_udi_rrwi()` and `mips_udi_i()` intrinsics.
  - The new <*mips/mips4ks.h*> header file describes 4KSc/d specific registers.
  - The <*mips/mips24k.h*> header file now lists the Config7 register bits.
  - The <*math.h*> header and mathc library no longer implements the `scalb()` function – the `ldexp()` function is equivalent and should be used instead. The new `ldexpf()` function implements the same function for single-precision float arguments.
  - The C library gains the C99 `strtof()`, `atoll()`, `llabs()` and `lldiv()` functions. The multibyte character support has been cleaned up, and the `strcoll()` and `strxrfm()` functions added to the library.

- More of the C library string handling routines have been tuned, and tested for strict conformance against the C99 standard.
- The <*sys/signal.h*> header file now defines the sig_atomic_t type.
- Building with `CRT0FLAGS=-DMINKIT` works more reliably, and produces smaller binaries.
- The Atlas and Malta real-time clock board support code now fixes the Y2K bug!

Release 5.03.06 Update

- Changes to this manual to improve clarity, update web download instructions, and describe use of new MIPSsim 4.x cycle counting facilities.
- The MIPS® Software Toolkit now includes full source code for MIPS Technologies' proprietary libraries: C (`libc`), maths (`libm`) and software floating point emulation (`libe`).
- When using the MIPSsim (`MSIM*`) board kits the example makefiles will no longer build ''ram'' and ''sa'' versions of an application, only the ''rom'' version is required for the MIPSsim simulator.
- The Malta board kits now include support for the SOC-it® system controller, as used in 24K core boards.
- The *sde-gprof* profiler will now ignore explicitly excluded functions (e.g. using **–P**) when calculating the scaling factor for the flat profile histogram.
- Allow use of 64-bit "long" and "paired single" floating point formats when MIP32 Release 2 ISA is selected (i.e. **–mips32r2**).

Release 5.03.05 Update

- Fixed gcc bad code generation for automatic const variable initialization.
- Fixed compiler crash caused by broken Cygwin 1.5.x mmap() system call.
- Made sdesetup.csh login script run across a wider range of Linux distributions.
- Added Insight ''View'' menu entries and shortcuts to open FS2 trace/trigger windows.
- Fixed FS2 trace line number / symbol name filter script to work on FAT file system.
- Profiling using MIPSsim simulator fixed.
- MIPS C/C++ intrinsics can now be safely used when compiling with −pedantic.
- All example programs now built with debugging enabled, for ease of usability with Insight (set NODEBUG=on to override).

Release 5.03.04 Update

- Minor updates to this manual.
- Several fixes to installation scripts and the new *mdi* command.
- MIPS intrinsics header files can now safely be used with the **–ansi** and **–pedantic** compiler options.
- All of the example programs are now built with debugging information (**–g**) enabled, so that they work better with the Insight GUI.
- The example makefiles' PROFILE option now enables profiling of C++ programs.

Release 5.03.03 Update

- Fixed MIPSsim profiling support in *sde−gdb*.
- Some clarifications in this manual.
- Minor packaging issues.

Release 5.03.02 Update

- New packaging – SDE is now just one component of the MIPS® Software Toolkit.
- Much simpler installation: now just one. or maybe two tarballs per host.
- Kit Improvements:
  - Removed support for non-MIPS Technologies targets and CPUs.
  - New run-time system for Malta and SEAD-2 boards using MDI i/o (i.e. via EJTAG).
  - Additional MIPS32 Release 2 intrinsics.
  - Added ISO C99 <*stdint.h*> and <*inttypes.h*>.
  - Support for 25Kf secondary cache.
  - Support multiple system controllers on Malta board (Galileo & Bonito64).
  - Determine CPU frequency dynamically on SEAD-2 board.
- GCC Improvements:
  - Adds support for the new 24K CPU core pipeline (**-mcpu=24k**).
  - Adds support for 64-bit floating-point unit on a 32-bit MIPS32 Release 2 CPU (**–mips32r2 –mfp64).**
  - Improved floating-point optimization for the 5Kf CPU (adds a 5Kf floating-point pipeline description).
  - Support branch-likely on 20Kc and 25Kf CPUs, but only when branch is ‘‘very likely’’.
  - Multilib hierarchy restructured: **–mips32r2** now gets its own set of libraries; **–mips16** and **–mips16e** are now subsidiary to the main 32-bit ISA, rather than a top-level ISA in their own right.
  - Support *gcov* profiling on MIPSsim simulator.
- GDB improvements:
  - Improved MDI remote file i/o.
  - MDI signal / exception handling added.
  - Now auto-generates a MIPSsim config file, if none is specified.
  - Supports MIPSsim instruction- or cycle-count profiling.
  - Handle cached/uncached address aliases.
  - Supports MIPSsim simulator version 4.x for 24K.
  - Fixes for 32-bit code running on a 64-bit CPU via MDI.
  - Improved support for FS2 EJTAG probe.
- Insight GDB GUI improvements:
  - Added help text for MDI targets.
  - Support MIPSsim cycle counting.
  - Improvements to Target Selection dialog – previous Target Name and settings now restored on first click of "Run".
  - Highlight stacked PC correctly in ‘‘Mixed’’ mode source windows.

Release 5.02.02 Update

A maintenance release, but with some significant changes:

- Earlier 5.x releases used an encoding in the ELF object file header for the the MIPS32 and MIPS64 ISA which was different from SDE 4.x. This incompatibility has been fixed, but you must recompile any object files or libraries which you previously compiled with SDE 5.0, 5.01 or 5.02 using the **–mips32** or **–mips64** options.
- SDE is now distributed under the terms of the MIPS Freedom-To-Use license. See the files `.../MIPS-FTU-USA` or `.../MIPS-FTU-INTERNATIONAL`.

- The Sparc version of SDE is no longer built for SunOS 4.x: it now runs only on Solaris 2.6 and above.
- The *sde-gcc* compiler has several bug fixes and improvements:
  - no longer crashes when reading a Windows/DOS (CR/LF) source file with an initial empty line, on a UNIX or Cygwin ''binmode'' file system;
  - fixes a bug which could generate bad code for soft-float;
  - load and store scheduling has been improved for dual-issue CPUs such as MIPS Technologies' *20Kc*;
  - now correctly distinguishes between zero-length and empty arrays as structure fields – zero-length arrays no longer generate an error, empty arrays can only be the last field;
  - will now use branch-likely instructions on *4Kc*, *5Kc* and *RC323xx* CPUs; the *20Kc* will only use branch-likely for branches which are predicted as very likely to be taken;
- The *sde-conv* and *edown* commands can now handle binary files in Cygwin ''textmode'' file systems correctly.
- The *sde-make* command can now handle Windows/DOS format text files in a Cygwin ''binmode'' file system.
- The GNU simulator can output a *gmon.out* pc-sample profiling file which can be merged with the call graph output from the SDE run-time system. The simulator now reports the presence of a floating-point unit and other ASEs via the MIPS32 Config registers.
- The *sde-gdb* debugger has many changes, including:
  - using a named MIPSsim configuration file no longer causes a crash on Windows;
  - remote debug protocols are much more reliable, resilient to errors, and interwork better with the Insight GUI;
  - inaccessible CPU registers will now be blank in the Insight register window;
  - on UNIX hosts you can use the mouse wheel to scroll Insight source window (if your X server is set up to support it);
  - now works well with the Abatron bdiGDB MIPS32 Ethernet EJTAG probe (ask Abatron about recent firmware updates);
  - downloading via TFTP to IDT/sim targets now works as advertised;
- The *sde-gprof* profiling tool now works.
- Accelerated versions of the `strcmp`, `strcpy`, `strlen` and `memcmp` functions have been added to the C library.
- The C library's *mcount* profiling code is now thread-safe.
- The on-chip timer support code now recognises the *20Kc* and explicitly enables the timer interrupts; it also adjusts for the different counter rate on the *20Kc*.
- An application can now be built with a ''minimal'' run-time system, omitting the stdio routines and POSIX emulation, by defining `CRT0FLAGS=-DMINKIT` in the application's *Makefile*.
- A CPU specific include file for the *20Kc* <*mips/ruby.h*> is supplied.
- The COP2, SmartMIPS and CorExtend (UDI) intrinsics have been improved.

Release 5.02 Update

- Bug fix release.
- Many improvements to this Guide.
- Product name changed to MIPS® SDE.

Release 5.01 Update

A moderate update to v5.0, but important in that it has a working Windows release.

In more detail:

- New instruction sets and extensions supported: MIPS32 Release 2, MIPS64 Release 2, the CorExtend ASE, the COP2 ASE.
- We now provide kits for MIPS Technologies' *Malta* and *SEAD-2* prototyping boards.
- Interface to MIPS Technologies' MIPSsim simulator (available to architecture and core licensees).
- The debugger's MDI interface has been expanded to provide target programs with file I/O (access to host's file system).
- The MIPS16e ASE can now be used on CPUs where the instruction memory is totally inaccessible to pc-relative loads by using the **–mno–data–in–code** compiler option.
- A number of board support kits for old boards are now no longer supported, although the source code is still supplied, see Appendix E "Unsupported Targets".

Release 5.0 Update

A substantial update internally; we've changed to a much more modern base compiler, and added support for important new CPUs and boards.

In more detail:

- The GNU compiler and other tools are now at the following revision levels:

binutils	2.9 (BFD 2.9)
gcc	2.96+
gdb	5.0
make	3.78.1

- The compiler includes the "Haifa" instruction scheduler for superscalar CPUs and implements the "DFA" pipeline description language.
- The MIPS-3D, SmartMIPS and MIPS16e ASEs are now supported.
- New CPUs supported: MIPS Technologies 5Kc, 5Kf, 4KSc, 20Kc CPU cores; NEC Vr5500.
- *sde–gdb* can use the MDI debug interface, giving it access to a range of CPU probes.

Appendix D: Key facts

File pathnames and tree of installation files

All these files start from wherever your installation started.

`COPYING`: Free Software Foundation's ''copyleft'' conditions.

`bin/`: binaries (specific to your release)

`doc/`: online documentation in PDF format.

`html/`: online documentation in HTML format.

`include/g++/`: C++ include files

`lib/gcc/`: ''hidden'' tools – compiler passes which get invoked by other programs.

`sde/bin/`: sub-programs invoked by the *sde−gcc* front-end.

`sde/examples/`: below here for the example files. Build your code like this (at least to start with).

`sde/examples/Makefile`: Build all the examples, one after another.

`sde/examples/hello/`: There are many more but each one is like this...

`sde/examples/hello/Makefile`: where *sde-make* starts. But works by including ''make.mk'', see below.

`sde/examples/hello/hello.c`: source files.

`sde/examples/dhrystone/`: more of the same.

`sde/examples/make.mk`: master make file for examples, which does most of the work of building a number of standalone forms of a program set out like the examples.

`sde/include/`: general include files.

`sde/include/mips/`: MIPS architecture-specific include files.

`sde/include/machine/`: synonym for the above.

`sde/include/sys/`: include files relating to POSIX system-call emulating library functions.

`sde/kit/`: where all the board kits live.

`sde/kit/Makefile`: master Makefile for the kit directories.

`sde/kit/MALTA32L/`: (example of many) low-level board support kit directories.

`sde/kit/MALTA32L/Makefile`: just sets SBD and invokes `kit.mk` from next-level up.

`sde/kit/MALTA32L/sbd.mk`: board support description file, for libraries or programs targetting a little-endian, 32-bit CPU on a Malta board.

`sde/kit/malta/*.c`: C source files specific to a Malta board.

`sde/kit/malta/*.sx`: assembler source files specific to a Malta board.

`sde/kit/free/MALTA32L.lib`: pre-compiled object file library

`kit/yamon/monitor.mk`: one for each monitor (`pmon`, `idtsim`, `yamon`, `mdimon`, `mtspmon`, `gnusim`, and the no-monitor `bare` option). This is included when building example programs for instructions as to how to link to a monitor PROM (or, in the case of `bare`, how to link without a monitor PROM).

`sde/kit/SKEL/`: starting points for a custom board support kit.

`sde/kit/kit.mk`: basic configurable makefile for building all board-support kits.

`sde/kit/rules.mk`: general compilation rules. included by just about all the makefiles, this ensures that *sde-make* always uses the compiler *sde-gcc*, and so on. It also adds support for the ''`.sx`'' suffix which we use to denote an assembler file which uses C preprocessor macros.

`sde/kit/share/`: files shared by two or more of the kit types.

`sde/lib/`: C, C++ maths and floating point libraries. Many variants of each library are available, as described in Section 11.3 “Multilibs”.

`sde/lib/ldscripts/`: control files for *sde−ld* − look at these if you need to change.

`share/`: files used the the *sde-gdb* graphical user interface.

Environment variables

Non-standard installations

The location of files and how to find them can be controlled by environment variables – essential to run the software in a non-standard place. You'll find information about that in Section 3.2 "Environment Variable Setup" above.

Makefiles

Other environment variables are used to configure program building for your particular target. The internal variables used by *sde-make* can be specified directly, or on the command line; but the makefiles also inherit the regular UNIX/Windows environment so you can also set up target configurations that way.

You've seen the first part of the following table before, in Table 9-2 "User-changeable "Make" variables for program building" above; but this is intended to be an exhaustive list of the mysterious variables met with when building SDE examples:

Variable Name	Default Value	Alternate Values	Description
ALL	`rom ram sa`	*any*	The default list of files to build.
ASFLAGS	`$(CFLAGS)`		Assembler flags.
CFLAGS	`-O2 -g`		C compiler flags.
CPPFLAGS			C pre-processor flags (e.g. `-D`, `-U`, `-A`, etc) to use when compiling the application source code.
CRT0FLAGS			Additional C pre-processor flags to customize the `crt0.o` startup code.
CXXFLAGS	`-O2 -g`		C++ compiler flags.
FLOAT	`no`	`yes` `ieee`	`no` = program doesn't use floating point; `yes` = some floating point support used; `ieee` = requests full IEEE–754 conformance (may increase program size significantly).
LDFLAGS			Additional linker flags.
LDSCRIPT		*any*	Custom linker script which overrides the standard one.
LDLIBS			Additional local libraries to link with program.
LIBC	`-lc`	`-lc -lm`	Library flags for compiler/linker. The examples master file `make.mk` normally works it out for you.
LIBCC		`-lstdc++`	Picks the C++ standard i/o stream and basic class library.
LOADLIBES			Additional standard libraries to link with program.
OBJS			Optional list of object files which make up the program.
PROG			Name of final executable file, see previous table.
RDEBUG	`no`	`yes` `immed`	Whether to include remote debug stub in program executable; see Section 14.4.2 "Serial Debugging with SDE Debug Stub". "immed" includes the stub and arranges to cause a breakpoint before calling `main()`.
SBD	`NOSBD`		Target board name, see Table 8-1 "Supported target boards and simulators"

Variable Name	Default Value	Alternate Values	Description
FEATURES			A list of run-time "features", separated by spaces, which you want included with or excluded from your application. For a full list see Table 9-2 "User-changeable "Make" variables for program building".
UNCACHED	no	yes	Whether to locate the program in cached or uncached space.
CPU		r4k r3k r5k	Set in the board-specific file $(SBDDIR)/sbd.mk, to select the basic CPU family. Can be overridden in the example application makefiles by APPCPU.
APPCPU	$(CPU)		Can be used to override CPU for application makefiles only, without affecting the board kit.
CPUTUNE	-mtune=$(CPU)	any	Can be set in sbd.mk to override the CPU tuning option.
CPUVARIANT		any	Set in sbd.mk to add additional CPU names to be asserted by the C preprocessor using #cpu assertions.
CP0	$(CPU)	m32 r3k r4k r5k	Can be set in sbd.mk to select the Coprocessor 0 support code, if it does not get set correctly by the default rules.
CACHES	$(CP0)	m32 r3k r4k r5k r54 rc32 rm7k cw01 cw10 lr30	Set in sbd.mk to a space-separated list of the cache architectures supported by all CPUs that could be fitted on this board (normally just one).
DBGSPEED	9600		Baud rate for remote debug serial link, used in board-specific serial-port driver (e.g. sbdser.sx). Set to any rate legal for your hardware.
DLFMT	s3	idt lsi relf	Download format to use, set in monitor.mk file and used to decide what kind of output file to produce when building examples.
DLSYMS		-y	Whether to include debug symbols in output file, set in monitor.mk.
ENDIAN	-EB	-EL	Build for a big-endian (-EB) or little-endian CPU (set in board-specific sbd.mk to match your CPU/board).

Variable Name	Default Value	Alternate Values	Description
FPU	no	yes maybe	Set in board-specific sbd.mk depending on your particular CPU: no = CPU doesn't have a floating point coprocessor; yes = CPU does have a floating point coprocessor; fp64 = 32-bit CPU has a 64-bit floating point coprocessor; maybe = probe for coprocessor at run-time.
ISA	-mips32	-mips64 -mips32r2 -mips64r2	Instruction set architecture to use (set in board-specific sbd.mk to match your CPU). Can be overridden in the example application makefiles by APPISA.
APPISA	$(ISA)		Can be used to override ISA for application makefiles only, without affecting the board kit.
KITDIR	.../kit		The "kit" directory which holds board-specific files, set in examples make.mk file.
KITFLAGS		-DXCPTSTACKTRACE *etc*	Set in sbd.mk to add C defines for a particular target board for use by only by other kit source files (not passed to application programs). The following may be useful for production builds, and/or to reduce the total ROM size: -DQUIETROM: Don't output any console messages from ROM startup -DSMALLROM: Don't include boot exception/error handlers
LAYOUT	rom	rom	Copy initialised data to RAM, but run code directly from ROM.
		romcopy, ram	Set in application makefile to control whether rommable code is linked to run in ROM, or copied to RAM.
MONITOR	bare	yamon mdimon mtspmon gnusim pmon idtsim	Selects what monitor PROM entry points are available to your program; "bare" implies that no monitor calls are used. Set by board-specific sbd.mk file.
SBDDIR		MALTA32L *etc*	Directory of kit files for your target board, relative to **$(KITDIR)**.
SBDDIRS	.		Where to find some kit files, starting at **$(KITDIR)/$(SBDDIR)**. Usually "." meaning right here, but different when many targets share one kit source directory.
SBDFLAGS		-DMALTA *etc*	Set in sbd.mk to add C defines for a particular target board for use by an "application" program.
SBDLOW **SBDOBJ** **SBDSRC**			Internal to board-specific sbd.mk, for building libraries. You need this only when changing the kit or building your own.

Variable Name	Default Value	Alternate Values	Description
FPFLAGS **RAMLDFLAGS** **ROMLDFLAGS** **RDBGFLAGS** **ROMDLFMT**			Internal to examples master file `make.mk` – you don't want to know

Appendix E: Unsupported Targets

These are still available to supported SDE customers, who may want to use them for inspiration.

SBD	Board Description	CPU	Endian
SKEL	Skeleton example		
ATMRT	LSI ATMizer R/T	L64360	B
BDMR4102/L	LSI BDMR4102	TR4102	B/L
COGENT	Cogent CMA101	R4300	B
GAL9B/L	Galileo G9	R4640	B/L
GAL9QB/L	Galileo G9 + A5230	RM5230	B/L
GSIM1B/L	GNU simulator, MIPS I code		B/L
GSIM4B/L	GNU simulator, MIPS IV code		B/L
GSIM16B/L	GNU simulator, MIPS16 code		B/L
GSIM16EB/L	GNU simulator, MIPS16e code		B/L
IDT134/L	IDT 79S134	RC32364	B/L
IDT332B/L	IDT 79S332	RC32332	B/L
IDT334B/L	IDT 79S334A	RC32334	B/L
IDT341/L	IDT 79S341	R3041	B/L
IDT355B/L	IDT 79S355	RC32355	B/L
IDT361/L	IDT 79S361	R36100	B/L
IDT364B/L	IDT 79S364	RC32364	B/L
IDT381/L	IDT 79S381	R30x1	B/L
IDT385/L	IDT 79S385	R30x1	B/L
IDT460B/L	IDT 79S460	R4x00	B/L
IDT465/L	IDT 79S465	R4640/50	B/L
IDT470/L		R4700,	B/L
		RC64474/5	B/L
IDT500/L		R5000	B/L
IDT575/L		RC64574/5	B/L
LSIPR	LSI Pocket Rocket	LR330x0	B
METEOR/L	LSI μMeteor	Tr4101	B/L
NEC41XX	NEC Vr41xx UEB	Vr4102	L
NEC4111	NEC Vr41xx UEB with MIPS16	Vr4111	L
NEC5074L	NEC DDB-Vr5074	Vr5000	L
NITRO/L	LSI Nitro	Cw401x	B/L
P4000B/L	Algorithmics P4000	R4400/4600/4700	B/L
P4000BSC		R4400SC	B
P4100B/L	Algorithmics P4032	R4100	B/L
P4300B/L		R4300	B/L
P4474B/L		RC64474	B/L
P4574B/L		RC64574	B/L
P4640B/L		R4640	B/L
P5230B/L		RM523x	B/L

SBD	Board Description	CPU	Endian
P5000B/L	Algorithmics P5064	R5000	B/L
P5260B/L		RM526x/7x	B/L
P7000B/L		RM7000	B/L
P6032B/L	Algorithmics P-6032	any	B/L
P6064B/L	Algorithmics P-6064	any	B/L
JALGOB/L		Jade	B/L
RACERX	LSI Racer/X	LR33020	B
SL3000	Algorithmics SL3000/ Radstone PME38−10	R3081	B
VME4000	Algorithmics VME4000	R4400	B

Appendix F: Document revision history

Revision	Date	Description
1.1	18th October 2004	First version with this title. Based on an original document first published by Algorithmics Ltd in 1995.
1.2	27th October 2004	Change `SDEMakefile` to `SDEmakefile`. Warn upgraders about the new **sde−insight** command. Note that inter-module optimization is not available with C++. Add new errata.
1.3	27th October 2004	Updated errata and change history for 6.01.01 release.
1.4	14th December 2004	New for 6.01.02 release: addition of 64-bit support and N32 ABI.
1.5	22nd March 2005	New for 6.02.00 release. Addition of MIPS16, MIPS DSP and MIPS MT ASEs. Board support kits for AMVP environment.
1.6	24th March 2005	Removed errata fixed in final 6.02.00 release.
1.7	29th March 2005	Reenabled −mcode−xonly compiler option.
1.8	21st April 2005	Improved MIPS16 support, including“`mips16`” and “`nomips16`” per-function attributes.
1.9	26th May 2005	Added MT debugging section; hardware watchpoints; extended “`set mdi asid`” command. Updated change history and errata for 6.02.02 release.
1.10	2nd June 2005	Added AMD-64 Linux to list of supported hosts.
1.11	27th June 2005	Added a section listing the compiler’s predefined macros. Updated change history for 6.02.03 release.
1.12	3rd October 2005	Updated change history and errata for 6.03.00 release.
1.13	12th May 2006	Updated change history and errata for 6.04.00 beta release.
1.14	31st May 2006	Added multi-VPE debugging section, and updated change history and errata for 6.04.00 release.
1.15	5th October 2006	Expanded the MT debugging section. Updated change history and errata for 6.05.00 release.
1.16	19th January 2007	Documented 74K core family and new DSP ASE revision 2 intrinsics. Described new SDEthreads API and TSP support. Added new board targets. Updated change history and errata for 6.06.00 release.